A Measurement and Simulation Methodology for Parallel Computing Performance Studies

by

Matthew Joseph Sottile

B.S., University of Oregon, 1999
M.S., Computer Science, University of Oregon, 2001

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Engineering

The University of New Mexico

Albuquerque, New Mexico

May 2006

Dedication

To my family.

Acknowledgments

The Los Alamos National Laboratory[1], in particular my manager, Ronald Minnich, allowed me time and resources to perform the work in this thesis. His advice and assistance was invaluable in the execution of this work, most often when my mind wandered down paths that deserved a dissertation of their own.

My summer student Vaddadi Chandu assisted in the creation of the Chama simulation prototype. Sung-Eun Choi and Erik Hendriks provided valuable feedback and assistance during the development of the FTQ microbenchmark.

The Department of Energy Mathematical, Information, and Computational Sciences group of the Office of Science funded this research. The ASCI Flash code used in this work was in part developed by the DOE-supported ASC / Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago.

The University of Oregon Neuroinformatics Center kindly provided parallel systems on which I performed many experimental studies.

Finally, I would like to thank my committee, Barney Maccabe, Ron Minnich, Bernard Moret, Jared Saia, and most of all, my advisor David Bader. David kept me on track and provided the positive guidance that got me through the process.

[1]Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36, LA-UR No. 06-2395.

A Measurement and Simulation Methodology for Parallel Computing Performance Studies

by

Matthew Joseph Sottile

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Engineering

The University of New Mexico

Albuquerque, New Mexico

May 2006

A Measurement and Simulation Methodology for Parallel Computing Performance Studies

by

Matthew Joseph Sottile

B.S., University of Oregon, 1999
M.S., Computer Science, University of Oregon, 2001
Ph.D., Engineering, University of New Mexico, 2006

Abstract

Disciplined application of system measurement and performance simulation is a powerful method for understanding the behavior of parallel programs and computers. The current state-of-the-art in parallel program performance analysis is focused on interconnection network and processor performance. The presence of operating system interference, although recognized as a source of performance degradation, has not been formally considered in the analysis process.

Distributed memory parallel programs often rely on periods of local computation that take the same amount of time to complete before synchronous communications are used to exchange data between processors. When the amount of time varies between processors, those that execute fastest are left idle while others catch up. This is particularly damaging to performance when the operations require a global synchronization where one slow processor can induce wasted, idle time on all others.

The hypothesis of this work is that a disciplined method of interference measurement coupled with a simulation of its effect on parallel programs can enable performance analysts to consider interference in their diagnosis and tuning process. This dissertation provides the following new results in this topic area that demonstrate the viability of this methodology:

1. The design, implementation, and application of the *fixed time quantum* microbenchmark for quantifying operating system perturbations on parallel computers is provided, representing the first detailed implementation and analysis scheme for such measurements.

2. A trace-driven simulation of distributed memory message passing programs is presented. This contributes to the field of performance analysis the inclusion of operating system perturbations as a parameter for performance simulation.

3. Performance sensitivity studies for several real-world parallel applications are given using the microbenchmarking and simulation tools.

This work shows that quantification of interference is possible through carefully constructed microbenchmarks, one of which has been demonstrated and is now in use by researchers in the field. The simulation tools developed to analyze the effect of this noise have extended the capability of existing analysis tools to integrate this new performance affecting parameter. Finally, this work demonstrates that the measurement and simulation methods can be applied to real codes to reason about their performance sensitivity.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The evolution of the parallel computing field has been unique in that unlike other areas of computing, it pushes the limits of capabilities and moves on immediately to new technologies before existing ones stabilize. This results in a highly diverse set of hardware platforms used in the progression of parallel computer architectures. At the coarsest level, there exist shared and distributed memory platforms, which have a significant impact on how programs are written to best utilize the computing resource. Within each memory model, one can further distinguish between massively parallel computers, clusters of commodity workstations, server class SMPs, and unique architectures of which on the order of dozens are ever manufactured. Within each, further variety exists with respect to processor architecture, from commodity microprocessors to highly novel designs such as the Cray MTA and IBM Cell, with all varying in their level of vector versus scalar capability.

Similarly, software used to program and control parallel platforms evolves rapidly and exhibits a large amount of diversity. Operating systems range from specialized,

"light-weight kernels" that offer virtually no features in the interest of minimizing performance impact, up to full fledged operating systems such as AIX and Linux. Networking protocols range from general purpose internet protocols through ones optimized for custom high-performance interconnection network hardware. Parallelism is programmed through a number of methods, such as explicit message passing, threading, or compiled high level languages. Each method has many alternatives in implementation. Explicit message passing can be achieved using the standard Message Passing Interface (MPI) library, the older Parallel Virtual Machine (PVM) library, or more esoteric libraries such as the Aggregate Remote Memory Copy Interface (ARMCI) library. A similar diversity exists for each other programming method.

One of the root causes of complexity in parallel computing derives from this extreme diversity in both hardware and software. A shared memory computer based on cache-free, massively multi-threaded MTA processors shares virtually no features in common with a cluster of Ethernet connected Pentium-class workstations. Regardless of this fact, the parallel computing community struggles to rectify the fact that these systems are treated as equivalent by users. As such, users who wish to run applications on parallel platforms find that application behavior varies widely between platforms, and varies differently depending on the structure of their application. An application developed on a shared memory SMP may run terribly on a commodity cluster, while an application that runs well on a basic cluster may perform poorly on a tightly coupled, massively parallel collection of relatively simple, slow processors (as realized in the recent IBM BG/L system). Virtually no tools exist for exploring both the causes of this performance discrepancy or the expectation a user can have of performance on a new platform given achieved performance on an existing system. At best, the parallel computing community has accumulated a large amount of folklore for system design and programming, along with a set of cumbersome tools that assist the user in mining through tremendous amounts of

parallel program performance data.

This approach is not appropriate for both application performance understanding and machine procurement and tuning. The limited set of applications and benchmarks used for machine comparison are representative of only a small set of applications, and give little to no indication of how a machine will perform under other workloads that will be faced in practice. Rarely are the actual applications that a machine is expected to execute in practice used as the evaluation and comparison tool at the time of purchase.

Similarly, application performance analysis is often an act of platform specific understanding and tuning. This renders codes of any substantial length difficult to port, dependent on a specific parallel programming model and runtime implementation, and difficult to maintain as the semantics of the parallel algorithm become lost in a growing set of platform specific tunings.

This dissertation presents a methodology for characterizing machine performance and the requirements and performance limiting factors for applications that fall outside the realm of traditional parallel benchmarks. The underlying theme of this work is that the entire process requires understanding of performance and behavior of both real platforms and applications under real workloads on real machines. Tasks related to tuning of hardware, operating systems, runtime layers, and application codes should not be performed independently, but in concert with each other following a disciplined, scientific process.

The reliance on idealized benchmarks as a primary means for comparing parallel platforms leads to performance discrepancies seen when parallel machines are acquired and extensive application tuning is required to better utilize the expensive computational resources that were purchased. Although valuable for characterizing a small set of specific applications, they disguise actual performance in general cases.

The methodology presented here asks the performance analyst to choose benchmarks and performance measurement tools that better span the space of performance relevant measures.

The proposed methodology is presented in three parts. First, the task of characterizing the performance characteristics of a parallel computer is addressed. Second, a method by which the performance characteristics of applications can be explored is presented through trace-driven application simulation. Each part compliments the other, and each is impossible to rigorously and completely understand in isolation. Thus both are required for a full, defensible analysis of parallel systems and programs. In recognition of this coupling of platform and application performance characteristics, a third portion of the methodology is proposed to merge both performance aspects within a framework in which a hypothesis can be posed and tested related to these characteristics.

1.2 Motivation

This work is motivated by trends and activities in the high performance computing field over the last decade. The days of "big iron" parallel platforms are dwindling as procurement decisions are driven by constrained budgets and managers who lack a coherent and accurate picture of the relationship of performance to specialized hardware. Clusters and cluster-like systems are here and will quite likely persist for the near future. Although the traditional parallel platforms were equivalently complex by component count and logical structure to modern clusters, they differed in that nearly all of the components of the system and software stack were provided by a single source. This source was capable of understanding the interplay between each component and could provide an interface to the application developer that was tuned to maximize performance.

Clusters of low-cost commodity components have lost this coherent design in the interest of favoring price over performance. The operating system is developed independently of the parallel programming support layer. Even worse, this operating system is not designed for high performance platforms at all. The parallel programming layer, most often in the form of a message passing layer such as MPI, PVM, or ARMCI, is developed independent of the interconnection network hardware. Efficient, stable parallel I/O is nearly non-existent. The suite of non-operating system tools, such as compilers and libraries, are developed by a community concerned with systems where performance is measured in terms of human interaction and usability instead of computational throughput. Finally, and most discouragingly, most of these components are connected together in an ad-hoc manner using scripting languages that are best suited to web pages and automation of tasks requiring text processing capabilities. Rube Goldberg would be proud of the contraption that is the modern parallel platform, although even he might flinch in the face of the gross lack of elegance in the engineering and design. The well engineered, elegant platforms such as novel Cray or IBM architectures are rare and expensive, frequently one (or few) of a kind. Good engineering has become an exception, no longer the rule, in the interest of cutting acquisition costs.

Given this picture, it is clear that there are many components that are forced to interact in ways that they may not have been designed to. They frequently are capable of such interaction on a purely functional level. Understanding this interaction with respect to its performance impact, particularly in a large-scale parallel system, is nearly entirely performed as an ad-hoc set of small scale measurements, feature specific analysis, each of which are performed independently with little to no cross pollination of knowledge.

Consider the following scenario. It is no longer uncommon to find parallel platforms composed of thousands of compute nodes, each populated with a small number

of processors, a block of memory, one or more interconnection network interfaces, and an array of other peripherals for I/O and monitoring. Each node also requires a minimally configured installation of a heavy-weight operating system, most often a UNIX-derivative that is predominantly developed to target the desktop or web services domain. Each node is quite capable of achieving reasonable performance for jobs that remain within the node, but performance of parallel programs requires a more careful look at the individual compute node and the effect local performance has on the global set of nodes.

It is widely accepted that the performance of parallel programs that require synchronous interaction between nodes is limited by the slowest node. In many cases, particularly within programs where the data set grows and changes over time, load balancing is required to ensure an even distribution of work to processors. Within a perfectly load balanced program, it is still possible that perturbations to the application due to the single node operating system or hardware can have a devastating effect on performance, as has been recently demonstrated [59]. Any time a local perturbation disturbs a process in the parallel program, that process is likely to become the "slow node", thus inducing idle time on the other nodes that are interacting with it while it catches up to them (see Figure 1.1). Since the operating systems and hardware installations on nodes are independent, then for a fixed period of time, as the node count increases the probability of a perturbation occurring during that period of time also increases. Due to the dependence on each node for the overall performance of the parallel program, rare events become more frequent, and low overhead interference on single nodes becomes strongly noticeable and intrusive.

In the case cited previously, the root cause of the performance problem was a misconfigured operating system on compute nodes. The procurement process for the machine required a great deal of benchmarking and measurement to ensure that it met the requirements outlined by application developers. Unfortunately, these

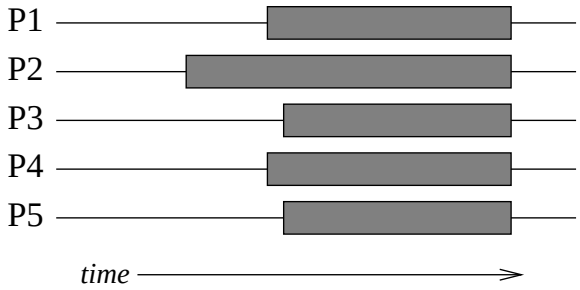

Figure 1.1: Illustration of skewed arrival times at collective operations. Shaded bar represents time spent at a collective, with $P5$ reaching it last and causing idle time on $P1-4$.

benchmarks only probed a strict subset of the performance characteristics of the machine, and the interference due to the operating system was not detected until the post-procurement performance tuning period. Furthermore, the application developers specified their requirements in fairly limited parameters such as bandwidth, latency, and memory requirements. They had no specification whatsoever regarding the sensitivity of their application to variations in these and other important factors.

This work proposes a methodology in which the performance measure and requirement set associated with both applications and machines is not limited to simple scalar values (such as runtime). Furthermore, this work will demonstrate that these performance characteristic vectors can also be used to perform performance sensitivity studies through simulation, allowing one to not only measure an instance of performance, but explore how it varies under different, real-world conditions within a parallel system.

1.3 Thesis Outline

This thesis examines the problem of characterizing both parallel platforms and programs, and describes how these separate descriptions can be unified to explore the sensitivity of parallel applications to performance dictating parameters. Chapter 2 provides an overview of existing work in this area that has taken place in the past and continues to the current day. In Chapter 3, the problem of describing machines is presented by examining the problem of deriving measures and building metrics for comparison.

At a level of abstraction above the hardware, Chapter 4 discusses tools known as microbenchmarks that can be used to infer characteristics that cannot be directly determined from hardware specifications and operating system parameters. The issue of operating system induced perturbations is discussed as an example of a non-trivial measure that can be empirically derived.

The work then addresses the problem of examining application sensitivity to these parameters. In Chapter 5, a trace-driven simulation is introduced. This simulation uses actual application traces from runs on specific parallel computers as input, and can be parameterized in order to explore the sensitivity of an application to variations in performance affecting parameters of the parallel computer.

In Chapter 6, this simulation is used to analyze a few real application codes. It is parameterized based on the data gathered with the microbenchmark from Chapter 4 and analysis is performed of performance effects due to operating system interference. Finally, in Chapter 7 the methodology is summarized and we discuss how it can be applied to enhance performance studies performed with existing tools.

Chapter 2

Existing Work

Performance analysis of both computer systems and programs is a very active area of research, and will continue to be such as long as new systems and programs are created and proposed. This thesis touches on two areas of performance analysis: measuring performance characteristics of machines, and applications that run on these machines.

2.1 Machine performance

It is virtually impossible to discuss sequential and parallel computers without encountering comparisons based on hardware performance metrics. These frequently examine characteristics such as processor clock rates, bus speeds, network latencies and bandwidths, and memory subsystem capacities and performance. These parameters have an unarguable importance in the performance of applications. All parallel applications are intrinsically built out of phases of sequential execution and communication of information to storage accessible to one or more participating processing elements. The characteristics of the machine are important, but not entirely, to

determining the performance of real applications.

2.1.1 Computational performance

All parallel applications at their core require computations to be performed on processing elements. Most scientific applications, particularly those in the physical sciences, require floating point computation. The historical metric for measuring performance is to look at the number of floating point operations per second (FLOPS) that can be achieved on mathematical operations common in scientific codes. This is predominantly measured through the execution of linear algebra operations and the fast Fourier transform. The LINPACK benchmark [19, 16] has been a long standing tool to measure the performance of large scale platforms required to solve large systems of linear equations. This is arguably one of the most important operations in codes from physics, chemistry, and engineering. The value of the LINPACK benchmark is undisputed in this respect, although its dominance in the field as the main metric for comparing systems fails to recognize that it is not the sole computational operation performed on parallel systems. It is interesting to note that, although it has been long recognized that computer performance evaluation should be performed in the context of real applications [33], the heavy reliance on idealized benchmarks such as LINPACK persists, over 40 years later.

In many fields such as plasma physics, signal analysis, and those that involve problems that can be solved with spectral methods, the fast Fourier transform [14] is an important operation. Parallel implementations of the FFT algorithm are very different from typical linear algebra benchmarks, and parallel benchmarking efforts such as Parkbench [29] include them in their code suite.

Integer performance is often measured, but presented as an aside to floating point performance. An increasing number of algorithms run on parallel computers

rely on purely integer-based operations. Traditionally, cryptographic and database applications were the primary source of integer based codes. While these continue to represent a significant portion of non-floating point computational workloads, an increasing number of applications from scientific computing are integer-based versus the traditional focus on floating-point based codes. Bioinformatics algorithms such as BLAST [2], graph analysis algorithms such as the DARPA HPCS SSCA#2 program [3], and many others exhibit complex performance behavior with solely integer-based computations.

2.1.2 Memory performance

Memory performance has traditionally been a key metric in comparing parallel systems, and was one of the major factors that motivated custom hardware in shared memory multiprocessors and massively parallel systems. Memory systems typically operate at speeds significantly slower than the processors that access them, and thus are a significant bottleneck for all but the smallest and most regularly structured codes. The introduction of the cache memory [35, 77, 66] in the Atlas and IBM System/360 Model 85 [38] computer showed that multi-level memory hierarchies can increase performance a great deal. The unfortunate side effect is that the time required to access memory is no longer uniform, and achieving the highest performance possible requires application developers to design their codes to take advantage of this structure.

Two metrics are important when examining memory performance. These are bandwidth and latency. Bandwidth determines what quantity of data can be moved from memory to the processor in a given unit of time. Latency determines the time required to move a single quantum of data from the memory to the processor.

The STREAM benchmark [41] has been traditionally used to measure the ac-

tual sustainable bandwidth that can be achieved by a wide variety of shared and distributed memory parallel, uniprocessor, and vector architectures. This bandwidth measure assumes regular access patterns and structure that is conducive to reaching near peak values, unlike the worst-case GUPS benchmark discussed below. STREAM has been also used to help compute the measure of "balance" that a platform provides, corresponding to the ratio of peak floating point operations to the peak sustainable memory bandwidth achieved [12]. A balanced machine is capable of moving data to and from the processor at a rate comparable to the processor speed, while the memory system within an unbalanced machine cannot keep up with the processor.

On the other end of the memory performance spectrum, the giga-updates-per-second (GUPS) benchmark exists to measure the performance of a synthetic kernel that performs a large number of small size, randomly distributed memory loads and stores [24]. This benchmark is intended to measure the global bandwidth that can be achieved within a system for worst-case single-word accesses from the global memory. This benchmark can also be applied to measure the worst-case memory latencies that can be expected of a system due to the speed of the memory access hardware and the structure of the memory hierarchy itself. It must be recognized that the result it produces represents a value that entangles effects contributed by the various levels of the memory hierarchy, most often the worst-case in which the higher performance cache levels are bypassed. To measure those individually, one would have to create a benchmark that has a memory traversal pattern structured for the specific hardware being measured. This would require either by explicitly coding this pattern for specific systems, or attempting to infer it in software in a manner similar to performance adaptive packages such as the Automatically Tuned Linear Algebra Subroutines (ATLAS) [76].

2.1.3 Communication performance

The communication fabric that binds disjoint computational elements and memory blocks together to form a single parallel computer is the final hardware characteristic discussed here. In modern high-end clusters, this is frequently one feature that can differentiate one system from another, even if the constituent nodes are identical. In the past, this aspect of hardware received more attention due to the expense of large scale interconnection networks. Given limited capacity switching hardware, a great deal of focus was given to the topology of the interconnection network. Various hardware systems and algorithmic techniques were created to perform well if the processors were connected in a high dimensional hypercube, torus, or mesh. With the proliferation of high performance networks outside the high performance computing field in areas driven by consumer and business applications, the price for high capacity, high node count fabrics has dropped. It is quite reasonable to purchase a full bisection bandwidth fiber switch for over a thousand nodes, making the interconnection network within a parallel computer essentially fully connected through novel topologies such as the Clos network [13] provided by modern Myrinet-based networks [53].

The focus has moved from the performance of exotic topologies and their corresponding routing algorithms, to the latency and bandwidth provided by modern fully-switched networks. The similarity in measuring memory systems and interconnection networks is no coincidence. Given a set of compute nodes, the programmer is provided not only a set of processors, but a set of distributed regions of memory. A parallel program performs computations on the individual nodes on both the memory attached to the node, and data stored within the memory of other nodes. Either via message passing operations or other remote addressing schemes, each processor uses the interconnection network to read and write to these distributed memories. In essence, the interconnection network introduces nothing more than another level

of memory hierarchy to the system. Similar methods to those for measuring traditional memory system performance can be applied to the interconnection network for measuring latency, bandwidth, and contention for resources.

2.2 Application performance

The parallel computing community has developed many tools and methods for analyzing the performance of parallel applications. Most often this is motivated by users finding themselves on specific platforms with a strictly limited allocation of time and resources. As such, analyzing the performance of an application is very important if they wish to maximize the utilization of their machine allocation and produce as many results as possible. The majority of tools in this area require performing real runs of applications, and analyzing the behavior of the application in a post-mortem fashion to identify regions that can best benefit from performance tuning and algorithmic restructuring. This post-mortem diagnostic method is valuable, but is based on an iterative approach of performing runs, performing analysis, and tuning the code. Less frequently, tools provide not only analysis of static run data, but methods for extrapolating expected parallel performance given either application or system level tuning. Little to no work can be found that combines system benchmarking with application analysis, as presented in this dissertation.

2.2.1 Parallel profiling

The concept of profiling is familiar to most programmers. On a sequential system, the GNU profiler (`gprof`) tool is well known and widely used. The program to be tuned is instrumented to emit data used to build its execution profile, and then executed. This instrumentation either occurs at compile time or at run time, depending on the

profiling tool and whether or not source code is available to the performance analyst. After execution, the profiling tool then can report to the user the percentage of time that was spent in each function call or finer-grained region of code, such as loops. This will give the programmer some indication of what region of code took the most time and would benefit most from performance tuning. This process of profiling, analysis, and tuning is repeated as many times as necessary until either the desired runtime is reached, or the effect and benefit of further tuning is not justified by the programming time and effort required.

Parallel profiling takes this one step further and allows the analyst to examine the amount of time each region of code takes on each processor while taking account the relationship between each processor with respect to message passing and synchronization activities. The popular and most mature tools for performing such parallel profiling activities are discussed here.

Tuning and Analysis Utilities (TAU)

The Tuning and Analysis Utilities (TAU) parallel profiling tool is a mature set of parallel program performance analysis tools that has been applied on very large scale platforms [40, 18]. Much of the focus of TAU has been on efficient generation of parallel program profiles and event traces on a wide variety of architectures. In addition to profile and trace generation, TAU provides sophisticated data mining tools for visualization and analysis of the parallel program behavior. This allows analysts to visually identify load imbalance, "hotspots" where optimization will be most beneficial, and relative behavior of separate threads and processors. The data produced by TAU is static and much of the analysis focuses on mining vast amounts of profile data to present it in many forms. It does not currently attempt to identify regions of interest to the performance analyst automatically, and performs little analysis or transformation on the profile data.

Vampir, Upshot

Unlike TAU, tools such as Vampir [75] (now the Intel Trace Toolkit) and Upshot [28] focus less on the infrastructure for gathering data, but on presenting it to the user in a reasonable fashion. Furthermore, where profiling tools focus on gathering local profiles for each parallel processor and relating them, Vampir and Upshot work primarily with data relating to the message passing behavior and processor interactions that occur within the parallel program. The visualizations they provide for aiding in analysis focus on the graphical representation of this interaction, which is used in this thesis work as the message passing graph concept that underlies performance sensitivity simulations. These tools are primarily used for visualization and data mining, and require the analyst to make decisions related to performance issues.

Pablo

The Pablo performance analysis environment [60, 61] is similar to TAU in its scope and goals. Pablo provided a suite of tools for program instrumentation, trace file formats, and data analysis. Like many performance analysis tools, the focus of Pablo was on data collection, mining and presentation. The literature on Pablo describes these tools, and provides information on trace collection and the performance perturbation it induces on the program being traced. This perturbation consideration is applied and discussed in this thesis with respect to the collection of traces used to drive our Chama analysis tool described in Chapter 5.

Dimemas/Paraver

The Dimemas [4, 25] and Paraver projects from CEBPA (Centro Europeo de Paralelismo de Barcelona) provide parallel program tracing, analysis, and simulation

tools for performance analysis of MPI and OpenMP-based parallel programs. This work is very similar to the work performed here in Chapter 5. In fact, this tool was explored as a possible vehicle in which to perform sensitivity studies by writing extensions to the existing Dimemas infrastructure. Unfortunately, the prohibitive node-locked license made it very difficult to obtain and execute in the cluster environments used for this work. Furthermore, the closed source nature of the tool made modification and experimentation with different perturbation techniques very difficult.

Dimemas primarily focuses on simulations of network effects on message passing programs, coupled with visualization tools for examining message passing structures within a parallel program. It does not contain any methods to evaluate the effect of on-processor, or on-node perturbations such as operating system interference on parallel program runtime.

Paradyn

The Paradyn suite [47] provides instrumentation, data collection, and analysis tools for examining the performance characteristics of large node count, potentially long running parallel jobs. This requires efficient data representation and collection methods, as a job that executes on thousands of processors for weeks will produce a tremendous volume of performance data. The "Dyninst" tool was created under the Paradyn project, and has found application in many contexts, including TAU. It allows runtime instrumentation of applications, avoiding the need to instrument source code before compilation. This is desirable in situations where source code is not available, or compilation requires more time and resources than users wish to spend.

2.2.2 Hardware behavior

Profiling tools most often present the user with a breakdown of how time is spent during program execution. This is not the only useful metric for describing the areas of interest for optimization within a parallel program. Nearly all modern microprocessors found in parallel computers provide what are known as hardware counters. These counters allow the user to monitor the number of low-level hardware events that occur over a period of time or range of instructions. These include details such as cache miss rates, TLB misses, branch prediction unit mis-predicts, pipeline stalls, and numerical operations completed. Given the performance degradation suffered when data must be accessed in main memory versus caches closer to the processor, it is often valuable to know not only what region of code took the most time, but that which required the most main memory accesses due to cache misses. Similarly, highly unstructured control flow may result in high numbers of branch prediction failures, reducing the utilization of the processor pipeline and the corresponding number of completed instructions per processor cycle.

The Performance API [9] (PAPI) is a mature API for accessing the hardware performance counters available on nearly all modern CPUs. It makes information about instruction scheduling (such as branch prediction statistics), cache behavior, and instruction throughput counts available to profiling tools. PAPI also allows multiple counters to be measured concurrently if supported by the hardware, allowing detailed and correlated analysis of multiple aspects of performance to be analyzed in a single run.

2.3 Methodologies

The recognition that most work in parallel systems and application performance analysis is ad-hoc at best is not new. Past researchers have sought to bring some sense of order and scientific discipline to the practice by proposing organized processes predominantly focused on system characterization and comparison. The practice of benchmarking new parallel systems, most notably for their placement in the "Top-500 ranking" [1], is a black art at best. A great deal of effort has been expended in tweaking and tuning the LINPACK benchmark to achieve maximal performance results. In the process, the benchmark number fails to capture anything more than the achieved performance. It does not take into account reliability and reproducibility, and the time and effort required to perform such tuning. Furthermore, it is not clear how to optimally run LINPACK on new, novel architectures such as the IBM Cell processor, BG/L, or Cray MTA. A system with a high rating may require a heroic amount of effort on the part of each and every application developer to achieve similarly high performance on their specific applications. Similarly, a system may score poorly due to the lack of a proper implementation of the code.

This was recognized in the late 1980s and early 1990s and efforts were started to bring some discipline to the benchmarking practice. These efforts defined sets of applications to be tested, and imposed strict requirements on the allowed tuning activities and the recording of the efforts required to perform them. The goal was to not only provide a peak performance metric, but a metric of the human factors related to achieving this performance, in addition to diversifying the set of benchmarks to provide a wider coverage of the set of codes executed. Although these activities have yet to show a clear impact (the Top500 rankings of the High Performance LINPACK code still dominate parallel computer comparison [16]), they mark a move in the community from ad-hoc comparisons to a more scientific discipline for performance analysis.

These efforts included application kernels such as the Livermore loops [43], the NAS Parallel Benchmarks [5], and the Perfect club [7]. Each sought to diversify the set of applications used for machine benchmarking to better match the workloads required by various institutions. The Parkbench effort [29, 30] was introduced to bring together smaller, similar efforts, and define not only a suite of applications and kernels for benchmarking, but a methodology for executing them and recording results. The primary purpose of this method was to make it easier for users to identify a benchmark application most similar to their own, and make more educated judgments with respect to performance comparison than by relying on LINPACK alone. Unfortunately, the Parkbench effort appears to have made little impact on bringing more discipline to machine comparison, as LINPACK persists today as the primary means by which parallel machines are compared.

Chapter 3

Machine characterization

3.1 Metrics and Measures

The need to compare two entities and generate a measure of relative similarity, and a quantitative assessment of relative quality, requires the definition of appropriate metrics and measures. This task of measurement and comparison is the basis of much analysis in systems performance, and drives many large-scale procurement decisions for acquiring new parallel computing platforms. Unfortunately, the connection between the platforms, the applications that will ultimately run on them, and the performance characterization benchmarks is ill-defined in the current state of the art. The connection relies on a loose notion that many scientific applications share common functionality (such as basic linear algebra operations) and structure. What is missing is an analysis of these characterization kernels and applications to determine whether or not they cover a sufficient amount of the space of performance influencing parameters, and to what degree the kernels themselves overlap in this respect.

To illustrate the problem of performance characterization, consider the mathe-

matical notion of a *metric* versus a *measure*. Given two entities (in our case, parallel platforms, applications, or characterization kernels), a *metric* provides a quantitative distance that can be used to compare, and ultimately understand the relative relationships between many entities. On the other hand, a *measure* simply provides a mapping from the set of measurable entities to the real numbers, with no necessary requirement that comparisons between these numbers say anything about the relative difference of the entities. Performance benchmarks are frequently constructed to provide measures, but rarely do they allow rigorous comparison between the measured entities on their own: they are not necessarily metrics.

Consider the following example. Two machines may be measured to have the same performance on a single simple application, such as a dense matrix-vector solver (ie, LINPACK), yet vastly different performance on random memory access applications (such as GUPS). This does not show that either benchmark alone is insufficient, but that neither is sufficient to fully characterize the space of performance characteristics that a particular machine exhibits. The dense matrix-vector measure may simply show that the projection of the vector of performance characteristics by the benchmark is identical - it does not show that projections via other operators will also be identical.

This introduces a concept that was used in this work to motivate the choice of applications used for benchmarking and analysis, and is similar to a proposed method for measuring productivity in high performance computing environments [67]. Every machine has a set of characteristic performance parameters that dictate how it will perform under different workloads. Similarly, applications have a set of performance parameters that dictate how they will perform on different machines. For discussion's sake, let us assume that k unique parameters exist for characterizing a machine. Therefore each machine can be described as a vector in a k dimensional vector space, with each individual parameter (component of the vector) corresponding to the basis

```
double epsilon = 0.0;
double x[1000];
double tmp;

for (i=1;i<100;i++) {
  x = f(x,1000);
  tmp = computeEps(x,1000);
  epsilon = AllReduce(tmp, MAX);
}
```

Figure 3.1: A simple kernel that alternates between phases of local computation and collective communication.

vectors that define the space.

We can further extend this concept of performance characteristics as defining a basis for a vector space by allowing benchmarks, performance measurement kernels, and applications, as projection operators that map the full set of components of the performance vector for a specific machine into a subspace of the overall performance space. For example, if one of the performance characteristics is the latency of the interconnection network, then a ping-pong test simply projects the characteristics of the machine onto the single basis vector that represents latency. A test of this form measures the time required to send a message of minimal size back and forth between two processors. By keeping the message size very small, a ping-pong test only measures the amount of time a single messaging quantum spends in flight between the two without sensitivity to bandwidth constraints.

More complex benchmarks (although seemingly trivial) perform projections that are a linear combination of more than one basis vector. Consider a simple benchmark that performs a collective `AllReduce` operation between phases of local computation, as shown in Fig. 3.1.

This seemingly simple kernel that forms the core of an algorithm such as one dimensional Gauss-Seidel or Jacobi iteration, touches on at least three performance characteristics. First, the array `x` is traversed by the functions `f()` followed by `computeEps()`, which stresses the local memory system. Both functions also are impacted by the floating point performance of the processing elements. The collective is impacted by interconnect latency (not bandwidth, as it is simply operating on scalars). Furthermore, the local node operations are impacted by a subtle performance characteristics commonly referred to as operating system noise or interference. During any phase of local computation, the program may be preempted by the operating system due to page faults, TLB misses, daemons, hardware interrupts, or other user-space tasks. This increases the amount of time required for the program to perform a fixed quantity of work. Unlike pure CPU floating point performance, or memory subsystem characteristics, OS interference introduces a new, stochastic parameter into the system that is subtle and difficult to quantify.

Thus we can see that a simple kernel projects the impact of many components of system performance into one or more measures. This is most frequently runtime, although other quantities such as cache miss statistics and instruction mix behavior are also measured. The key observation is that a projection to a space that is too low dimensional (such as a scalar) causes a great deal of information to be irretrievably lost regarding each performance characteristic of the machine and it's interaction with the computational kernel. Metrics based on these low-dimensional projections can be misleading or simply provide little information of interest.

3.2 Performance vectors

Typically a benchmark gives a single scalar, which is a measure of the machine and benchmark, not necessarily usable as a metric. As such, we consider performance

quantification as vectors of distinct characteristics. Defining what the elements of the vectors represent requires examining the aspects of parallel programs where performance analysts have traditionally identified sources of performance sensitivity. These can be broken down into many characteristics, some of the most common being:

- Variability in arrival time at collective operations
- Blocking on message passing operations
- Synchronization primitives
- Memory latency and bandwidth
- Floating point and integer performance
- I/O

Instead of measuring scalar values for codes, we instead should measure these characteristics, and populate a performance vector. The scalar traditionally used, such as runtime, can be considered the result of a vector norm on these performance vectors. The HPC Challenge benchmark [17] defines seven benchmark kernels that step beyond the basic LINPACK measurement. These form an initial attempt by the HPC community to define benchmark kernels such that a vector description of machines can be created. Although it is not known to what degree these kernels cover the full set of performance characteristics, they represent a step in a positive direction away from traditional single scalar value metrics for machine characterization.

This approach opens up the possibility of novel ways of exploring performance data to reason about the relationships of applications to the underlying machine, and similarities between applications themselves. One approach is to apply principal components analysis (PCA) to the data. The motivation behind PCA is to take the

high dimensional performance vector data, and identify the components containing the most variation. By limiting the number of principal components that are considered, we can reduce the dimensionality of the data onto the performance aspects that show the greatest diversity in performance.

The use of different benchmark kernels for quantifying performance vectors will allow comparisons that not only provide scalar differences for comparing machines (such as peak floating point throughput differences), but allow one to reason about similarities and differences between machines considering multiple performance dimensions. Ideally a set of performance characteristics can be identified that form a basis for the space of performance metrics that are derived from and covered by a small set of kernels. The HPC Challenge benchmark suite represents progress in this direction by the community.

3.3 Operating system perturbations

In Chapter 4, the specific performance parameter related to quantifying operating system perturbation is discussed. This performance parameter is commonly referred to as OS perturbation, OS interference, or OS noise. Its basis is in the widespread use of preemptive time-sharing operating systems for high performance computing systems, most often some form of UNIX derivative such as IRIX, Linux, or AIX. This performance characteristic can form part of a larger performance vector to take into account the preemption characteristics and rates of various platforms. In these systems, user applications coexist on computers with operating system daemons and kernel-level processes. These processes and daemons related to operating system tasks require periodic slices of time on resources shared with the user application. As such, the user application periodically is preempted for these activities. During these periods of preemption, the application performs less usable work than a period

of equal duration in which no preemption occurs.

Operating system interference is a problem because, although largely transparent on desktop workstations or small processor count jobs, it can have a strong negative impact on large scale parallel job performance. Parallel applications frequently require synchronous operations of some form, ranging from basic blocking send and receive operations, to more complex collective operations involving sets of processes. If each process in a parallel program performs essentially equivalent amounts of local work before reaching one of these synchronous calls, then the amount of time spent idle on each process is small since they will all reach it at approximately the same time. If the operating system causes one or more processes to be preempted, then these processes will reach the synchronous region later than processes that were not preempted. The overall efficiency of the program is thus reduced because the time spent idle (with no usable work being completed) across the parallel set of processes is increased.

Operating system interference forms only one element of the performance vector that we seek to construct to describe a parallel system. Given that the number of possible components of these vectors is very large, instead of attempting to cover each with a high level discussion, we instead focus a great deal of attention on the measurement and quantification of one. This rigor should represent the amount of attention each and every component of the performance vector deserves.

3.4 Application vectors

Performance vectors are not confined to platforms alone, but can be generated for applications. Often application performance requirements are discussed in broad terms, either discussing the sort of operations it relies on (such as collectives), or presenting an ad-hoc estimate of the granularity of computational work that occurs

between communication operations. One can make a more rigorous vector description of an application by analyzing the behavior it exhibits at runtime. The simulator presented in Chapter 5 can also serve as a statistical analyzer simply as a side effect of the method used to reconstruct the message-passing graph of the application.

The vector description of an application can not only contain simple metrics such as the memory requirement or instruction mix, but details such as the proportion of each type of message-passing operation. For example, an application vector could contain the number of processors a run was performed on and the percentage of calls that were `MPI_Allreduce` versus `MPI_Bcast` collectives on each processor. For each, the vector could also be populated with the mean and variance of the duration every processor spent blocked in each event type, allowing the vector to capture the load imbalance experienced by the program in a very rough sense. Examples of these statistics are provided in Chapter 6 for some of the codes that were examined in the form of MPI operation mix and average exclusive times spent in each event type.

Chapter 4

Microbenchmarking

The concept of a microbenchmark, or synthetic benchmark, for probing specific aspects of system performance has existed in the performance analysis literature for nearly four decades. A program of this form is intended to probe the system in order to reveal a specific performance characteristic such as bus bandwidth or the cache structure. Ferrari [21] differentiates between microbenchmarks (which he refers to as internally driven tools), and external tools that probe the system without consuming system resources. A key point that he points out, that motivates much of the work in this chapter, is that self-interference is impossible to avoid for the internally driven tool due to it sharing resources with that which it measures. Taking that into consideration, this chapter presents a carefully constructed and analyzed microbenchmark for quantitatively measuring operating system perturbations as observed by user-space programs.

This chapter extends the work described in [69] published at the Cluster 2004 conference entitled "Analysis of microbenchmarks for the performance tuning of clusters," and provides more depth for the description of the microbenchmark code and aspects of the analysis of the microbenchmark time series data.

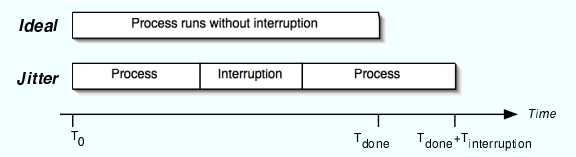

Figure 4.1: Interference and how it can slow an application.

4.1 Introduction and Background

Cluster-based supercomputers continue to be a popular high-performance computing platform, and have recently broken into the large-scale supercomputing realm previously occupied by custom and traditional parallel architectures. The most immature component of clusters relative to traditional HPC platforms is the compute node operating system architecture. Until recently, compute nodes were no different than workstations in both hardware and software configuration. With minimal modification to Linux itself, systems such as BProc [27] and OpenMOSIX [11] are able to eliminate much of the on-node software that is not necessarily used on compute nodes. This second generation of cluster node software architectures eliminates obvious sources of perturbation: daemons that require resources, unnecessary kernel code (such as unused modules), and excessive hardware. The primary motivation for eliminating hardware is one of reliability, but a secondary benefit is that it also eliminates potential sources of noise due to hardware interrupts.

Noise due to hardware interrupts or other interference can lead to a program running in a longer time that without. We show a simple situation in Figure 4.1.

In the first case, the program is fine – it starts, runs with the whole CPU, and ends at the expected time, T_{done}. In the second case, some interrupt has occurred, caused by either hardware or software, and the program completion time is delayed by the time consumed by the interruption, $T_{interruption}$.

Characterization of single-node performance is vital in understanding the performance of large scale parallel computing platforms, especially clusters. Most non-trivial parallel programs alternate between phases of disjoint computation on each processing element, and phases of communication and synchronization involving all or some of the processors in the system. Ideally each processor participating in a synchronous operation will reach the operation at the same time as each other participant. If this is not the case, each processor that arrives at the operation before others must wait until the slower processors arrive. This idle time decreases utilization of the system, and has a negative impact on runtime. This notion that the slowest processor dictates the overall performance is well known in parallel computing folklore. Traditionally this is addressed when examining load balancing to minimize wasted cycles and overall runtime. We examine this issue because perturbations induced by factors outside application codes, even those that are themselves perfectly load balanced, can cause individual processors to be slowed down, thus causing idle and wasted cycles. As node counts in modern clusters increase into the thousands, the cumulative effect of even minute perturbations on individual nodes can collectively result in a tremendous impact on application performance.

The reason for this impact is rooted in an unfortunate combination of large node counts and independence of perturbation events across nodes. The operating system on a cluster does not synchronize activities such as daemon wake up, hardware timer interrupts, and so on. Anectdotal evidence for spontaneous self-synchronization has been observed, but as a lucky and unintentional side effect of operating system scheduling on individual nodes. For a single node, the rate of such interrupts can range from fractional Hz for daemons and kernel tasks, to over 100Hz for hardware activity. If the only perturbing events occur at a rate of kHz, where $k < 1$, a single processor activity clearly achieves a high utilization on the processor. For n nodes, the likelihood at any given time of a perturbation of at least one node rises to $n * k$.

Scientific applications have very special characteristics relative to the suite of applications a general operating system is designed to support. First, consider the simple uniprocessor case. A simple application such as a 2D grid-based solver requires long periods of sustained computation on a fixed amount of memory. In a well written implementation, locality can be leveraged to use caching or vector hardware to perform long sequences of simple operations on many data elements very quickly. Caching in particular requires that in order for these long sequences to occur quickly, the cache remains consistent and does not need to be read again. Similarly, the processor should not switch contexts and give cycles to other processes. If memory, TLB, cache, or any other time critical component is corrupted by another process, there is overhead involved in reloading elements so that the instruction sequence can complete. If a process is preempted for other activities, there is both time spent by the other activities and that for context switching. Both are high overhead and increase the amount of time a given amount of work takes to complete.

Locating sources of interference can be difficult. Once the obvious sources of interference such as daemons are removed and compute nodes are turned into essentially dedicated computational resources, the investigation begins. Many portions of the Linux kernel have sufficient performance for general use, but tuning may be required to minimize the impact of portions such as the scheduler or software interrupts on both compute-heavy and globally synchronous workloads characteristic of many scientific applications. Furthermore, in distributed systems such as BProc, there are infrequent but periodic activities that the operating system performs to ensure that the overall cluster remains functional over time.

A key activity in locating interference is the quality of the measurement tools. Over the last 10 years researchers have developed several tools to measure interference, usually as a set of microbenchmarks [20, 44, 72]. A popular microbenchmark is the 'fixed work quantum' (FWQ) benchmark, in which a program reads the time,

performs a fixed amount of work, reads the time again, and saves the delta of elapsed wall-clock time. The FWQ has shown its value several times for determining that interference exists.

As it happens, the FWQ benchmark has limitations. It fails to take into account important fundamentals of sampling theory [57, 56], and it can easily lead to misleading conclusions. For example, a common analysis technique is to look at the variance of the time it takes to perform the work. On a saturated processor the variance measure will show no interference at all – it is not possible to distinguish between a very slow processor and one with no interference. This is discussed in detail in Section 4.7.

Finally, and most importantly, the data output of FWQ is unusable with many popular techniques, such as the Fourier transform and periodogram analysis [8], due to the fact that samples are taken every quantum of work. Reconstructing the amount of work performed as a function of time is very difficult, and requires assumptions about the distribution of work over the sampled time interval that are unverifiable. As such, it is difficult to make statements about activities in the system in either the time or frequency domain given the lack of temporal data.

In this chapter we discuss the flaws of FWQ benchmark, and provide a new microbenchmark, *fixed time quantum*, or FTQ, which yields far more meaningful results. We then show some of the results from FTQ and how it can be used for frequency domain analysis of interference. Since many of the causes of interference are periodic in nature, frequency domain analysis can point to a root cause of interference.

This work is motivated by the following condition. We examine the amount of work a program is able to do by periodically sampling the work it has achieved. The clock we use is accurate enough to characterize many different phenomena in both the operating system and the hardware. The microbenchmark we have developed

guarantees that we adhere to good practice in that our sample intervals are fixed in duration, occurring at periodic intervals, and that the samples are taken on well-defined time boundaries. We can measure both utilization as percent-of-peak and interference. Preemption of processes and inefficient cache usage cause a lower utilization of a processor for a given workload. Our goal is to increase (not necessarily optimally) the utilization achieved by applications by tuning the operating system to reduce high impact interference and overhead.

We present our measurement and benchmarking methodology, experimental setup, data analysis, and preliminary conclusions. The benchmark this work is based on is detailed in Sections 4.2 through 4.5. We provide a detailed description of the experimental setup (both software and hardware) and process (how and when measurements were taken) in Section 4.6. Finally, Section 4.7 provides a basic overview of the analysis techniques we have and are continuing to investigate to identify features in the raw benchmark output that are related to noise sources. We also show data resulting from this study that illustrates the relevance of this work.

4.2 Microbenchmarks

In this section, we begin with a discussion of timers that are used for reporting timing data for microbenchmarks. We then discuss an existing, widely used microbenchmark based on timing fixed workloads and the flaws inherent in its design. Given these flaws, we present a microbenchmark that overcomes these flaws by measuring work completed in a fixed quantum of time.

4.2.1 A brief discussion of time

Before we discuss microbenchmarks that use timers, we need to discuss timers. The most common timer that has been used to implement benchmarks of this sort is `gettimeofday()`.

Consider an inherent limitation of software timers. If t units of real time elapse from some measurement t_0 and its successor t_1, we have to assume that there is some small error ϵ that forces

$$t = (t_1 - t_0) + \epsilon \tag{4.1}$$

If we wish to derive information about the distribution of the interval lengths in a series of time samples, we must minimize or eliminate ϵ by carefully choosing the `readtimer()` implementation. Our benchmark code currently uses a processor tick-based timing function, which is known to be accurate to within a sub-nanosecond resolution. Many benchmarks that are similar to this one use the more common `gettimeofday()` function from C. That timer has a higher amount of variance and a lower precision than counting raw processor ticks, making it difficult to assess the effect of fine-grain hardware or operating system activities.

Another consideration is the Nyquist frequency and sampling theorem. In sampling-based systems, the Nyquist frequency is twice the highest frequency which can sensibly be measured for a given sampling rate. The sampling theorem states that reconstruction of a band-limited signal (i.e., the power spectrum for frequencies $\omega > B$ is zero), one must sample at a rate $\omega_{sample} \geq 2B$. If $\omega_{sample} = \omega_{Nyquist} = 2B$, the signal is said to be *Nyquist sampled*. In a system with a 2GHz clock, to fully reconstruct a signal band-limited to $0 \leq \omega \leq 200Hz$, sampling at a Nyquist frequency of 400Hz is achievable with the high resolution timers. If a microbenchmark uses the `gettimeofday()` clock which runs at 100Hz, then we can only really talk about

signals of 50Hz or less. High resolution timers driven by the processor clock can support sampling with a Nyquist frequency of nearly 1GHz. As such, they allow microbenchmarks to detect perturbations occurring at realistic frequencies of 100-200Hz, or interrupts occurring at several KHz.

Most modern processors provide some mechanism to measure processor ticks for high resolution timing applications. The Pentium, PowerPC, AMD Athlon, Sparc, Alpha, and MIPS processor families all have provided cycle-based counters for performance analysis for many years. For robust performance analysis and measurement, one should not use counters with poor precision and accuracy such as `gettimeofday()`. One cannot assume that the underlying implementation uses high-precision processor timers, so it should simply be avoided. In our experiments, we use the Pentium Time Stamp Counter (TSC). Furthermore, many timers provided for libraries such as MPI (e.g., `MPI_Wtime()`) use `gettimeofday()`. Understanding the capabilities and limitations of the available timers is vital to reaching meaningful and sound conclusions from time measurements. It should be noted that the FFTW [22] package includes a self-contained header file (`cycle.h`) that can be used to supply cycle-accurate timer access on a wide variety of architectures and operating systems.

4.2.2 The Fixed Work Quantum (FWQ) microbenchmark

The FWQ benchmark is used extensively in the community. It measures the amount of time required to perform a fixed amount of work, where each unit of work is either a memory reference, floating point, or integer arithmetic operation. The code can be summarized in pseudocode as shown in Fig. 4.2.

Although trivial at first glance, there are features of this that must be understood to properly interpret data. First, to preserve the raw data, we must store it in memory. Given that n is assumed to be significantly larger than a cache line size,

```
for i=1 to n do
  timestamp = readtimer()
  timeseries[i] = timestamp
  for (j=1;j<workcount;j++)
    operation
  end
end
```

Figure 4.2: Pseudocode for the Fixed Work Quantum (FWQ) microbenchmark.

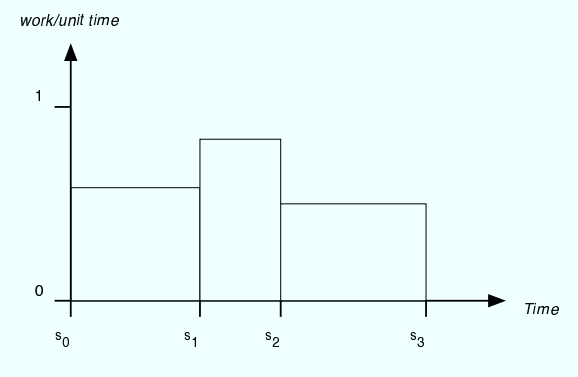

Figure 4.3: An example of three samples of the FWQ benchmark.

we will see some caching effects when the index addresses a location not located in cache close to the processor. Modern systems with predictive prefetching and other hardware optimizations will be able to take advantage of the simplicity of the workload to hide these effects, but in general they will be present to some degree.

For high performance nodes, with complex sources of interference, the FWQ benchmark does not work well at all. It will provide some indication of interference. We have seen that it is just as likely to provide no indication when interference is maximized and uniformly distributed over the sampling period.

Finally, there may be a huge amount of interference, which the benchmark might reveal in some gross sense, but the data will not reveal the details. This is true regardless of the quality of the timer used. The more interference there is in the system, the more interference in the benchmark and the more variance in the actual timestamps. The sample intervals will thus be of variable width and variable starting point. They are useless for any type of signal processing. This is illustrated in Fig. 4.3 where three samples are taken of a fixed workload. The fixed workload corresponds to the area of the rectangle over the time period required for the fixed amount of work to complete. The FWQ benchmark only dictates this constant amount of work with no information regarding the distribution of this work over fixed width time intervals.

The data, rather than revealing variance, is a mirror of the variance of the system. It is as though the system were sampling noise instead of information. Here we see a fundamental flaw in the fixed work quantum benchmark.

4.2.3 The Fixed Time Quantum (FTQ) microbenchmark

In response to the shortcomings of the FWQ benchmark we developed the fixed time quantum (FTQ) benchmark. FTQ is well-behaved in the way it samples and hence can be used for frequency- and time-domain analysis. We show the inner loop that performs the sampling in Fig. 4.4.

The basic operation is simple. The loop will count work done until it reaches a fixed end-point in time. It then records the starting point of the loop and the amount of work that was done.

This microbenchmark measures *work per unit of time*, with a fixed unit of time, ending at a periodic interval. It does everything right that the other microbenchmark does wrong. The data from this benchmark can hence be used with such tools as

```
1  terminal = 1 << 21;
2
3  while (!endloop) {
4    last = readtimer();
5    endinterval = (last + terminal) & (~(terminal-1));
6
7    for(now = last, count = 0; now  < endinterval;) {
8      count += work_unit_result();
9      now = readtimer();
10   }
11
12   totalcount[(done)*2] = last;
13   totalcount[(done)*2+1] = count;
14   done++;
15   if (done > dototalcount)
16     endloop = true;
17 }
```

Figure 4.4: Pseudocode for the Fixed Time Quantum (FTQ) microbenchmark.

the Fourier transform and formal time series analysis methods.

The difference between measuring work per unit time versus time per unit work is the key to the analysis being able to accurately identify periodic features in the data. Periodicity of interference is measured in cycles per unit time, most commonly cycles per second (Hz). Periodicity in the FWQ benchmark data is cycles per unit work, which is very difficult to correlate with activities that are periodic in time. We briefly looked at methods for interpolating time per unit work data into work per unit time, but gross assumptions have to be made about the distribution of work over the measured time period for each sample. Our desire to use tested techniques from the signal processing and spectral analysis fields led us to find a way to sample the system in a domain that those methods are well defined over: fixed time interval

sampling.

FTQ is specifically constructed to ensure well disciplined sampling is performed, making not only statistical analysis possible, but opening the door for more sophisticated time series and spectral techniques to be applied to the data. The lack of proper sampling in previous, similar microbenchmarking efforts, makes such analysis mathematically difficult, or more often, impossible.

The code presented in Fig. 4.4 is structured to illustrate the algorithmic structure required to perform the disciplined sampling that we sought to present. The core concept is to perform many units of computational work that required significantly less time per unit than the desired sampling interval. By checking a high resolution timer at the end of each work unit, we can attempt to make samples begin and end on time boundaries that are within small error bars of the desired sampling interval. The subtle detail is that, due to the perturbations that FTQ is actually attempting to measure, it is entirely possible that the microbenchmark is perturbed at the end of a sampling interval, making a sample extend farther than desired into the following interval. A simple computation of the end of the subsequent sample is required to compensate for this error in order to make future samples obey the same sampling time axis as those prior to the perturbation.

In the remainder of this chapter, we examine other important aspects of the benchmark in order to tune it to continue to obey proper sampling practices, while making operating system or hardware induced perturbations clear and easily exposed within the data. We must define the work that we wish to observe being perturbed and understand the impact of its structure with *and* without the presence of perturbations on the resulting sample sequences. We will also present details regarding techniques for FTQ data analysis.

4.3 FTQ Sampling

We designed the FTQ benchmark to overcome issues in other benchmarking methods that sampled the amount of time required to perform a fixed quantum of work. Sampling to examine temporal events generally requires that the sequence of samples represent measurements of some quantity over a fixed time interval known as the *sampling interval*. The sampling interval must be kept reasonably consistent in length for each sample in order to infer temporal features of the system being sampled. When the sampling interval is defined by a fixed quantum of work, with the samples themselves representing the time required to complete the work quantum, the regularity of the sampling intervals with respect to time is lost. FTQ fixes this by enforcing a strict time-based sampling interval, and measuring the amount of work that can be achieved in that interval.

The method by which FTQ fixes the time interval is subtle due to the fact that the microbenchmark is a self sampling system that must not only measure the quantity that is being sampled (work), but also monitor the sampling time so as to determine when a sample has been completed and another is to be started. FTQ achieves this in the following manner, and relies on the presence of a timer that has a resolution significantly higher than the desired sampling interval length. In a modern microprocessor, high-resolution timers are available that are cycle-accurate, or work at a time resolution on a similar scale as the cycle timings. Consider a 2GHz CPU, which has a cycle time of $2 * 10^{-9}$ seconds, or 0.5ns. If we wish to sample at intervals of approximately 1ms, then 2,000,000 cycles will occur between samples. How do we achieve this in FTQ? Lines 1 and 5 in the code shown in Fig. 4.4 are the key.

Assuming that we wish to sample at approximately 1ms intervals, corresponding to 2,000,000 cycles, we recognize that this is approximately 2^{21} = 2,097,152. During each sample interval only the lower 21 bits of the timer value will change. As

soon as the 22nd bit changes value, we know that 2^{21} cycles have occurred, and a new sampling interval has started. The first line of the FTQ code creates a binary integer, `terminal`, with a single one bit set in the 22nd bit position, representing the sampling interval. The fifth line is required to take the time at which a sample starts, and determine the timer value that will be considered to be the end of the current sampling interval.

On this line of code, two important operations are performed. On the previous line, the current value of the timer has been read into the variable `last`. The sub-expression `last + terminal` computes the value of the timer that is precisely `terminal` timer units in the future from the time represented by `last`. The second sub-expression is required to enforce a strict sampling boundary for all samples, regardless of the value of `last` where a sample starts. This is to deal with potential perturbations that will cause a sample to start at some point inside the desired sampling interval. This subexpression, $\sim$`(terminal-1)`, represents all bits at positions equal to or greater than the exponent in the sampling interval. `terminal-1` contains bits set in positions 21 and below, so the bitwise NOT sets all bits in positions 22 and above to one. The combination of the sub-expressions with a bit-wise AND operation creates a value stored in `endinterval` that represents the end of the sampling interval that is on a strict sampling interval that ends no more than 2^{21} timer units in the future.

The reason this is performed instead of simply setting
`endinterval = last + terminal` is that it is inevitable that the value of `last` does not fall on a clean time value with all lower 21 bits set to zero. Once a sample has been taken and the loop between lines 7 and 10 has exited when the timer has hit or exceeded one of these clean sample end times, there are operations that occur that store data and check to see if more samples are required. By the time the outer loop reaches line 4 again, the timer is no longer at the value it had on line 9 in

the most recent sample. In fact, it is entirely possible that a perturbation due to process preemption during execution of lines 11 through 16, and the loop test on line 3, resulting in their execution taking more than 2^{21} timer units, thus making it necessary to freshly sample the timer on line 4, and re-compute the end time for the next sample based on this current value.

This has two side effects. The first is highly desirable. Each sample is guaranteed to span a sampling interval that is some multiple of 2^{21} timer units plus some $\delta < 2^{21}$. Ideally this multiple is one and $\delta \approx 0$. In the presence of noise though, the second side effect is possible, where this multiple has a value of 1 or more. In this case, there may exist samples that have intervals that exceed the desired interval. Fortunately, these are not only dealt with as they occur so as to not pollute subsequent samples, but by saving the timer values in addition to the amount of work performed (line 8), they are detectable and can be dealt with cleanly in subsequent analysis of the raw data.

By performing the operations implemented on line 1 and 5 in the code, FTQ achieves a regular temporal sampling pattern, and yields data that not only represents a clean time series of work-quanta per sample, but a time series of sample boundary times that can be used to rectify perturbations to the FTQ self-sampling scheme.

4.4 The FTQ work quantum

In this section, the FTQ code shown in Fig. 4.4 is examined focusing on line 8. Consider the two time series in Fig. 4.5. These were taken on an IBM X41 laptop running Linux before the operating system `init` process has started daemon processes. The FTQ used to probe the system in the case on the left-hand side used a unit of work that consisted of a single integer increment operation, `count++`.

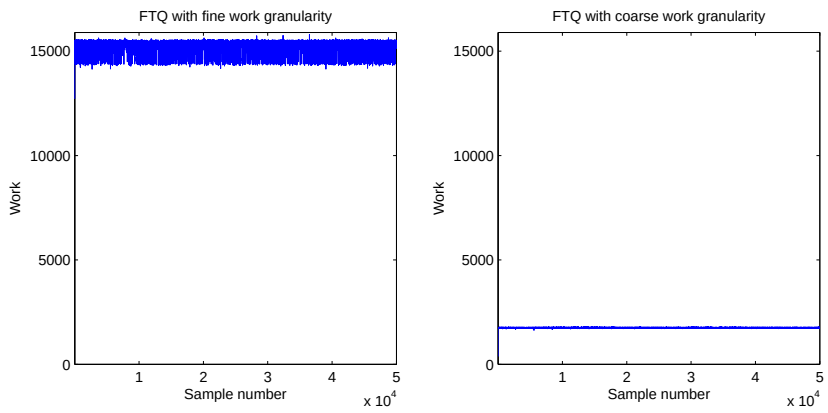

Figure 4.5: FTQ data taken at boot time. Left plot based on fine grained work quantum, right plot based on coarse work quantum.

```
for (k=0;k<ITERCOUNT;k++)
  count++;
for (k=0;k<(ITERCOUNT-1);k++)
  count--;
```

Figure 4.6: A coarser FTQ sampling work quantum that can be unrolled to (`ITERCOUNT`*2)-1 integer operations.

On the right-hand side of Fig. 4.5, we show the FTQ time series at the same point during booting, but with a coarser work quantum. This work quantum consists of a sequence of integer increment and decrement operations that have the same overall effect on the `count` variable as illustrated in Fig. 4.6. In the case shown, the value of `ITERCOUNT` is 32 with the loops completely unrolled, yielding a work quantum composed of 63 integer operations.

In this example, operating system daemons and other processes have been removed from the experimental setup, and the data shows FTQ as the sole process running on the node aside from the operating system kernel. The data is shown on

an identical work axis to illustrate that the effect of increasing the work quantum granularity reduces variance, but also reduces the number of samples that can be achieved over a single sample period. In this case the sample period corresponded with 2^{20} processor cycles in both cases. We will now examine the basis for the difference in the data observed with varying work quanta, and the reason why this must be considered in performing experiments to remove low-level hardware effects that are not of interest in the study.

4.4.1 Work unit granularity effects

The effect due to the granularity of the FTQ sampling work unit is clear from the data, and second in importance only to the algorithmic structure that enforces strict sampling boundaries. Tuning the granularity of this workload is dependent on understanding what causes the variations in samples due to the granularity, and finding the best granularity for the platform being examined. The root of the difficulty in understanding the expected behavior of the code versus what is observed is the underlying architecture. It is impossible to expect a modern microprocessor to execute code in precisely the order in which it is specified in the presence of out-of-order execution methods, branch-prediction and speculative execution, and pipelining. Amazingly enough, pipelined architectures, although not realized seriously in hardware until the IBM STRETCH and CDC 6600, were first alluded to (although indirectly) in the work of Burks, Goldstine, and von Neumann nearly 60 years ago [10].

Consider the simple, fine grain work unit where a single counter is incremented. Embedded within the larger sampling loop, the majority of the instructions that occur during each sample are actually related to timer maintenance, sample interval adjustments, and data storage. This code is largely branch oriented, requires writing to L1 data cache (at best), and requires computations for conditional operations.

```
int x = 66;
int endloop = 0;

while (endloop = 0) {
  x += 33;  // work unit
  if (x > 777)
    endloop = 1;
}
```

Figure 4.7: C code implementing an analogue of the coarsened FTQ work unit.

On the other hand, the work unit is code that, in our current investigations, should be solely using functional units operating on data that can reside inside registers, to remove the performance effects of the memory subsystem and, potentially, the speculative execution units within the processor.

As the granularity of the work unit is increased, the proportion of work instructions to sampling instructions moves work instructions to the majority of those present in a single sample, making the perturbations due to process preemption more obvious and not overwhelmed by CPU and memory related effects of the process on itself.

Consider a very simple portion of code that embodies the structure of the FTQ sampling loop with the work unit indicated by a comment, shown in Fig. 4.7.

On the PowerPC G4 microprocessor, GCC emits 3 lines of assembly to implement the work unit, while 14 lines of assembly are required for the work unit embedded in the simple sampling loop. This is shown in Fig. 4.8. Note that lines corresponding to labels are left out of the line count. The unusual constants for the work increment versus stopping criteria are intended to make the corresponding assembly statements easily identifiable.

```
//
// Work unit alone
//
  lwz r2,28(r30)
  addi r0,r2,33
  stw r0,28(r30)


//
// Work unit embedded in sampling loop
//
  b L7
L3:
  lwz r2,28(r30)
  addi r0,r2,33
  stw r0,28(r30)
  lwz r0,28(r30)
  cmpwi cr7,r0,777
  ble cr7,L2
  li r0,1
  stw r0,24(r30)
L2:
L7:
  li r0,0
  stw r0,24(r30)
  lwz r0,24(r30)
  cmpwi cr7,r0,0
  bne cr7,L3
```

Figure 4.8: Unoptimized PowerPC assembly code for C code shown in fig. 4.7 as generated by GCC 4.0.

This implies that $\frac{11}{14}$, or 78.6% of the instructions executed for each sample are not directly related to the work unit itself. In the actual FTQ code, with the simple `count++` work unit, the proportion of non-work instructions is far higher. Clearly the loops in Fig. 4.6 are trivially unrolled to straight-line code with no control flow structure, thus it represents a viable coarse grained work quantum. It can be tuned by changing the value of `ITERCOUNT` at compilation time to achieve the desired proportion of work to non-work related instructions per sample.

4.5 Self-perturbations of FTQ

Clearly the structure of FTQ itself induces perturbations that appear in the FTQ samples, and must be accounted for when analyzing the data. We must be able to understand these self-induced perturbations so that they can be recognized in the data to differentiate their effects from those that we are actually looking for, such as operating system or hardware interrupt driven process preemption. A cursory examination of the FTQ pseudo-code from listing 4.4 reveals that, in addition to the loop structure and work unit, there is code for storing samples for later storage. Performing I/O to disk or other non-volatile storage during the core of FTQ is obviously undesirable, thus the data is stored in memory until the sampling has completed at which time it can be saved without impacting the sampled data itself.

As FTQ becomes complete in our description, we should consider graphically the activity that occurs during a single sample interval in order to understand what precise set of activities (and possible self perturbations) are included in each and every sample. This is illustrated in Fig. 4.5. A single sample contains multiple timer reads, work unit executions, end interval adjustments, and two writes to sample sequence storage. All of these other than the writes to sample storage can be considered to be operations solely on data stored in CPU registers, and the overall instruction

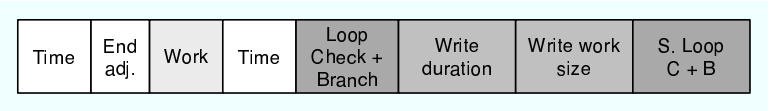

Figure 4.9: Activity of FTQ during a single sample

stream can be considered sufficiently small such that its execution does not induce L1 instruction cache misses. To validate this, we show cache miss statistics for the full FTQ program versus the main sampling loop in Chapter 6.

4.6 Experimental Methodology

Accurate benchmarking is a difficult process, and frequently conclusions that are reached are not as strong as they could be due to under-specified and poorly controlled experimental conditions. This section discusses the environment in which experiments with FTQ were conducted in the interest of making similar experiments easier for others to execute and compare. We also point out potential variables that are not currently controlled during the experiment, and must be considered in whatever conclusions are drawn from the data.

4.6.1 Hardware and software configuration

The objective of the experiment is as follows. Given a compute node, evaluate the behavior of a fixed benchmark in the presence of different operating system environments. Identical binaries are used regardless of the operating system configuration. The hardware remains constant - no devices are to be removed or added. In cases where devices are unnecessary, such as network-based booting of nodes, the drives required for other operating systems were left in place and enabled. A later experi-

ment will then test whether or not unused devices being removed has any influence on the outcome of the benchmark results. In this chapter, the data we present was gathered on a single operating system configuration from an existing BProc-based cluster node. We are now at the point where the benchmark is tuned and understood to the point where we can begin to do cross-operating system comparisons.

The benchmark code itself is described in detail in Section 4.2. Before running the benchmark, the compute nodes are rebooted and the benchmark is immediately executed on a fresh node. The code will be executed for an identical sampling period, and results will be returned to the front-end node when the code has completed. To eliminate artifacts in the data due to poor timer quality and accuracy, we chose to measure durations in terms of processor ticks, using the Pentium TSC. The TSC is a cycle counter, implemented as a register, and is hence accurate to within several hundred picoseconds. This accuracy is in stark contrast to the time of day clock, which is only accurate to several tens of milliseconds.

The kernel on the compute nodes is the Linux SMP kernel version 2.4.25 patched to include BProc. The compute node is composed of a dual processor motherboard with 2.20Ghz Intel Xeon CPUs.

Another variable to consider is the actual code generated by the compiler versus that in the benchmark C code. Careful use of volatiles can ensure that no optimization occurs in critical segments. Also, for each run, optimization settings must be the same. All of the code discussed in this chapter was compiled with all optimization options turned off.

4.6.2 Processor scheduling issues

The hardware configuration used in this experiment is an unmodified node from a moderate-scale (128 node) cluster. As such, the node was configured with both CPUs

active on boot. Without the ability to set processor affinity for specific processes, it currently is not possible to guarantee that for a given run of the benchmark the execution occurs completely on a single processor. If the code is scheduled to the other processor during execution, we expect to see a performance hit due to cache effects, internal scheduling within the processor (instruction level), and other related issues. We hope that instruction level issues within the CPU occur fast enough to be beyond the granularity of the timer we are using, but the current work here does not quantify or take this into account. We also would like to run similar experiments with the second CPU disabled at boot time to explore the effect of multiple processors.

4.6.3 Generating a baseline measurement

It is not possible to avoid system-related artifacts in the FTQ data stream, such as cache activity and hardware interrupts (such as clocks). Fortunately, these are present regardless of the operating system configuration, thus it is possible to derive a baseline measurement of the system itself. Using this baseline, we can then have some confidence of decoupling the system-generated noise from that induced by the operating system. How to measure this baseline though is non-trivial. We propose two methods, one of which is used in this chapter to generate results, while the other gives the most "raw" picture of the machine being benchmarked.

The first method, that is used here, requires only an operating system that is capable of booting a custom `init` program. The scripts used to build this are inspired by those used by the "bootchart" project that attempts to profile the booting process of various operating systems. The key to this method is that the `init` process is the first process started when the kernel boots. As such, a process started in its place will have a guarantee that no other processes will be competing for resources, thus the only perturbations at that time are due to the hardware and basic kernel itself.

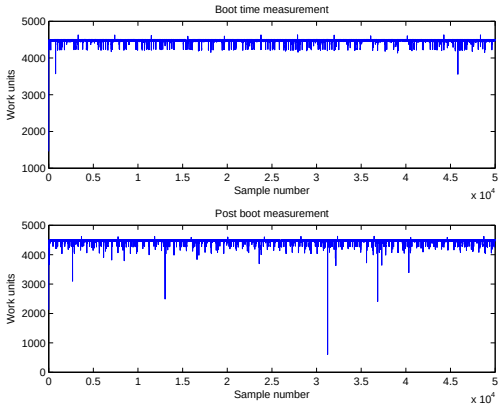

Figure 4.10: FTQ raw data at boot and postboot sampling times.

The second method allows an even lower-level measurement to be made, ensuring that the operating system kernel, in addition to all processes, are non-existent during the measurement. In this case, we boot the machine directly into the microbenchmark, instead of bootstrapping the microbenchmark with a full operating system. This provides the cleanest measurement of the system, but can result in settings to devices being different than those present when an operating system is booted. This can easily be accomplished using LinuxBIOS [49] to boot the benchmark.

In this chapter, we show results using the first method. Fig. 4.10 shows the results of running FTQ at boot time, followed by an identical run after the system has booted. The system used for the test was an IBM X24 laptop running Debian Linux version 3.1 with the 2.6.11 kernel patched with `perfctrs` to allow for profiling of low level hardware behavior. The post-boot FTQ measurements were taken with the GDM display manager started, but without a login into the system under the graphical desktop software such as GNOME. The FTQ work unit in this case was composed of 25 integer arithmetic operations (13 increment, 12 decrement).

In addition to the raw data being clearly cleaner at boot time, we can also see

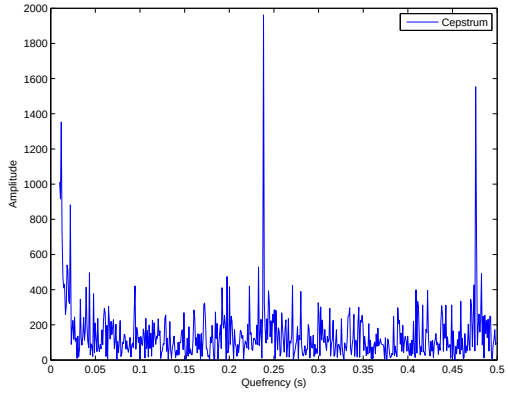

Figure 4.11: FTQ cepstrum at boot time

that the periodic components of the frequency spectrum of each data set has become more diverse after booting. In Fig. 4.11, we see the cepstrum during boot time, while Fig. 4.12 shows the cepstrum once booting has completed and all daemons and other processes have started.

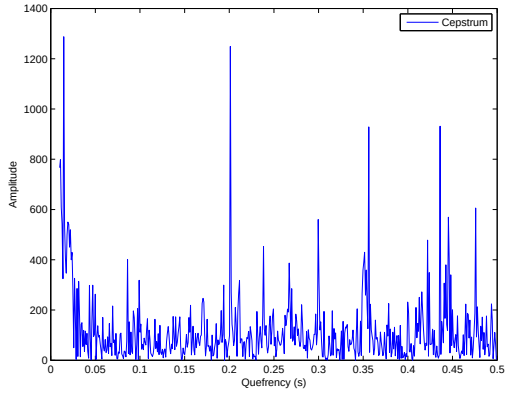

Figure 4.12: FTQ cepstrum post boot

4.7 FTQ Data Analysis

The analysis phase of this investigation is performed with raw data from the FTQ microbenchmark as described earlier. Qualitative analysis of the data makes it clear that simple visualizations (such as scatterplots) reveal structure and pattern in the data, versus what could appear to be random or nearly featureless results. Unfortunately, qualitative techniques are only able to state that there are in fact features in the data that are of interest. The goal is to then quantitatively identify these features by any of a wide variety of techniques.

The first techniques we will discuss are basic statistical methods for analyzing time-series data. These are necessary to answer important questions that impact the choice of more sophisticated methods, such as whether or not the series are stationary or if correlations can be found in the series at different time scales. Calculation of metrics such as the mean and variance are necessary to understand the distribution of measurements. Unfortunately, basic statistics and plots (such as histograms) have an intrinsic problem due to discarding of temporal relationships in the raw data points. Two very different time series (one with periodicity, one without) can yield identical sample statistics (mean, variance) and histogram plots. As such, we have and continue to explore more sophisticated techniques that take this into account. To illustrate the little amount of information contained in simple techniques such as histogram analysis, consider Fig. 4.13 showing a histogram of the first-order differenced FTQ data discussed in detail later. As we can see, most of the data indicates no perturbations between samples, with the majority of samples containing only a small perturbation, and very few outliers. Furthermore, as we will see later, this plot contains no indication of the rich structure of the data in the frequency domain.

Considering that kernel bookkeeping, system heartbeat daemons, and hardware

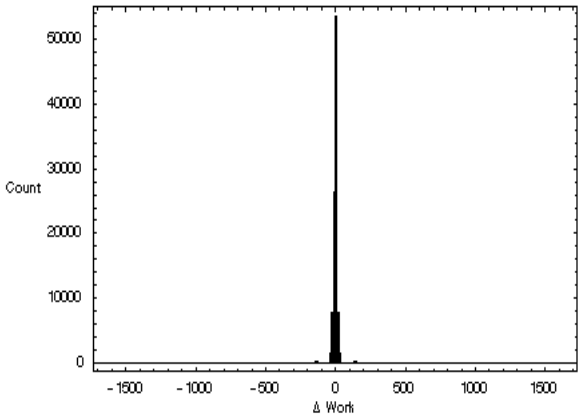

Figure 4.13: Histogram of first-order differenced FTQ data.

interrupt timers are all periodic in nature, we have been exploring the use of spectral methods to extract periodic signals in the data. This work is still in the exploratory stage and no concrete algorithms have been created that are able to, given a raw benchmark output time-series, provide a set of frequency components that represent periodic interference due to software and hardware.

4.7.1 Results

Fig. 4.14 presents the raw data produced using the FTQ benchmark. The data contains a clear peak-workload at 18077 units of work, with a number of samples that performed less units of work in the fixed sampling interval. The samples are predominantly near or at the peak workload, as the mean of the data is 18072.6 units of work, with a standard deviation of 39.16 units over 100000 samples. Many of the non-peak data points show an obvious periodicity, and this motivates our exploration of spectral methods next.

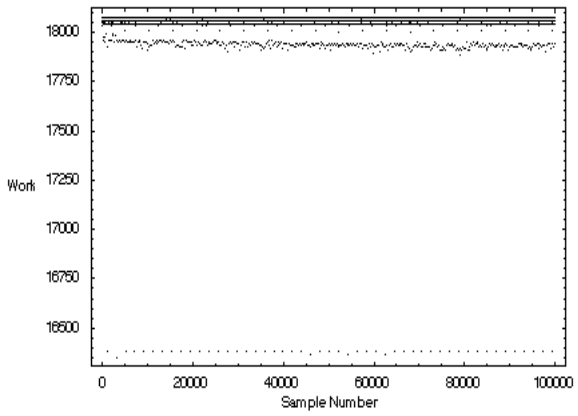

Figure 4.14: Plot of raw data for the FTQ benchmark.

4.7.2 Identifying periodicity

Our original and primary goal remains to identify noise in the system due to periodic activities that are not part of user application code. Theoreticians and practitioners in time series analysis have attacked this problem for many years for problems ranging from population dynamics to sunspot cycles in solar data. Although their techniques are far from perfect (consider the complexity of stock market or climate data), for simple systems such as ours, they are quite capable. The basis of our current work is examining features in the frequency spectra of the sampled data using the discrete Fourier transform. A deep explanation of this technique is outside the context of this paper, but the main idea behind it is that most signals can be approximated by composing a set of sinusoidal signals with varying frequencies and magnitudes corresponding to the periodic features of the original signal. More information can be found in [57] and [56].

Consider the sequence of samples shown in Fig. 4.14. For ease of analysis, first order differencing of the data can produce a zero-mean stationary sample sequence $\hat{S}$ from the original sequence of N samples without loss of frequency-domain infor-

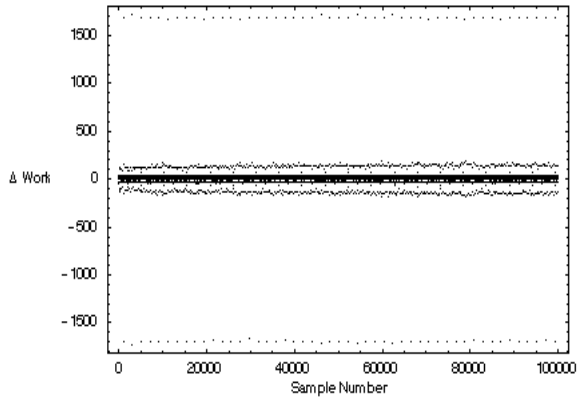

Figure 4.15: Plot of data for the FTQ benchmark after first-order differencing.

mation by defining:

$$\hat{s_i} \in \hat{S} = s_i - s_{i-1} \ , \ i = 2 \ ... \ N \ , \ s_i, s_{i-1} \in S \tag{4.2}$$

The resulting sequence $\hat{S}$ is shown in Fig. 4.15. Using the Mathematica `Fourier` function, we produced the spectrum shown in Fig. 4.16 based on 2000 consecutive samples.

As we can plainly see, the frequency spectrum contains a small number of obvious dominant frequency components. In the future we would like to perform experiments to isolate these periodic sources of noise and correlate them with hardware or operating system sources. It is nontrivial to immediately decide what is causing each instance of noise by simply examining the dominant spectral components. The microbenchmark has an upper bound on the highest frequency signal it can reliably measure. Features that are at or above this limit may appear aliased in the measured data set, making it difficult to distinguish them from reasonable lower frequency sources.

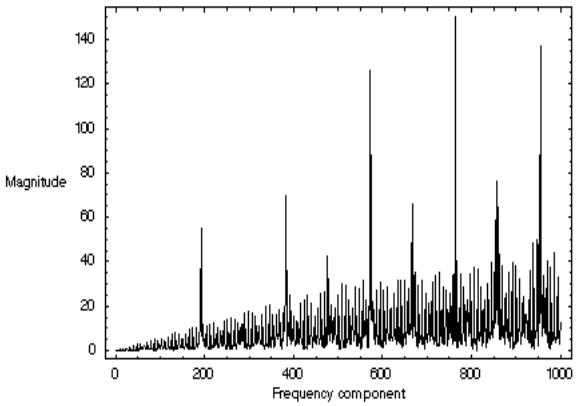

Figure 4.16: Plot of the output from Mathematica `Fourier[]` function for 2000 consecutive samples.

A set of experiments can be performed where a single node is configured, individual software and hardware components removed, and measurements taken of the interference in the absence of one or more different components. Each modification of the node would allow the spectrum from the modified configuration to be compared to the original, fully configured node, making it possible to identify which spectral components (if any) disappear or change when the component is removed. These spectral components can then be correlated with specific parts of the machine hardware or software configuration.

Once we do this work, we will be able to relate the magnitude of frequency components in this graph to their impact on the application, for example we should be able to state that a frequency component that is only a few dB above the noise floor will have little impact, whereas one that is 20dB over the noise floor may have larger impact. For the first time, we can sample a system, process it using signal processing techniques, and draw conclusions directly from the data.

4.7.3 Statistical ambiguity

We have already stated that basic statistics, such as the mean and variance, can be misleading or devoid of information when the source signal does in fact contain information of interest. Furthermore, visualizations such as histograms that are commonly used to qualitatively analyze benchmark data are in fact a very poor tool for identifying correlations between samples due to periodicity and temporal trends.

The use of the variance and mean of a time-series of benchmark data is common, but one must be very careful not to draw complex conclusions about the source data from these simple metrics. Consider a situation where a benchmark is run twice, once on a system with no noise, and once on a system with constant noise (i.e., a constant perturbation uniformly distributed over the time-series). Let δ be the perturbation that interferes with each sample, $S = (s_1, s_2, \cdots)$ be the sample without noise, and $\hat{S} = (\hat{s}_1, \hat{s}_2, \cdots)$ be that with noise. Since the noise is uniformly distributed over the sample sequence, we can let $\hat{s}_i = s_i - \delta$. Assuming the length of S and $\hat{S}$ is N, their means are defined as

$$\mu = \sum_{i=1}^{N} \frac{s_i}{N} \tag{4.3}$$

$$\hat{\mu} = \sum_{i=1}^{N} \frac{(s_i - \delta)}{N} = \left(\sum_{i=1}^{N} \frac{s_i}{N} \right) - \delta = \mu - \delta. \tag{4.4}$$

Similarly, the variance of S and $\hat{S}$ can be shown to be equal, regardless of δ.

$$\sigma^2 = \sum_{i=1}^{N} \frac{(s_i - \mu)^2}{N - 1} \tag{4.5}$$

$$\hat{\sigma}^2 = \sum_{i=1}^{N} \frac{(\hat{s}_i - \hat{\mu})^2}{N - 1}$$

$$
\begin{aligned}
&= \sum_{i=1}^{N} \frac{(s_i - \delta - (\mu - \delta))^2}{N-1} \\
&= \sum_{i=1}^{N} \frac{(s_i - \mu)^2}{N-1} = \sigma^2.
\end{aligned} \tag{4.6}
$$

As such, basic statistical metrics are of little use applied to benchmark data, including both the FTQ and FWQ methods presented earlier. To truly extract non-trivial information from a time-series, more sophisticated techniques are required.

Finally, it is frequently the case that time-series data is distilled down to a single visualization that collapses the time axis out of the sequence. One such method seen in practice is the histogram. Fig. 4.17 shows that two signals with very different periodic structure yield the same histogram.

A periodic signal may in fact produce a visually interesting histogram, but the structure of the histogram easily can be shown to have little or no correlation with periodic structure in the source data. Furthermore, the histogram can in fact strip away all interesting information other than a visual representation of μ and σ. As an example, consider the zero-mean first order differenced data in histogram form shown in Fig. 4.13.

4.8 Data analysis revisited

In this section, we revisit the analysis aspect of the FTQ microbenchmark, and discuss how one can take raw FTQ data and begin to extract measures and metrics for the system on which the benchmark was executed. These can then be applied to guide performance studies of applications and cross-machine comparisons.

First, recall the following formal definition of the sampling intervals employed

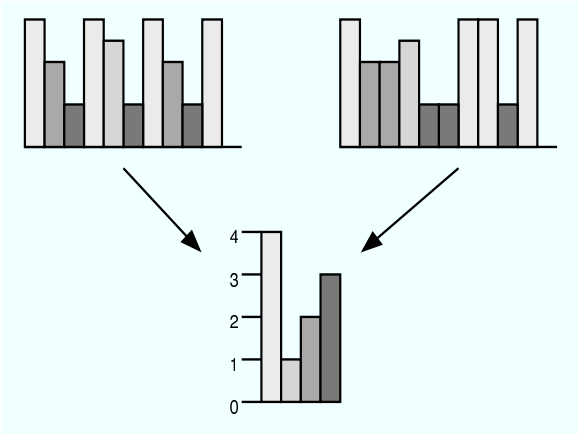

Figure 4.17: Two different time-series with an identical histogram. Shading of histogram "buckets" corresponds to shading of samples.

by FTQ and the sample data itself. If we consider the sampling period to be T_s and the kth sample spanned a time of $T_s + \delta$ ($\delta < T_s$), then the subsequent sample $k + 1$ will sample in a window of $T_s - \delta$. Given that the amount of work chosen to execute during sampling requires significantly less time than a single sampling period ($T_s >> T_{work}$), the value of δ should not exceed T_{work} very often. One final issue needs to be dealt with to provide a data set of high integrity in the time domain. It is very possible that a heavy-weight perturbation may occur during sampling, resulting in a single small quantum of work to take not only more than T_{work}, but T_s itself. Although the subsequent sample will be adjusted to return to the proper sampling interval, this sample will have an anomalous duration. No method of sampling can overcome this. By recording not only the amount of work performed per sample, but the time stamps for the sampling boundaries, we can identify these unusual samples and deal with them accordingly during analysis.

Traditionally microbenchmark data analysis is an exercise in basic statistics and probability, with a rare excursion into uncertainty quantification and frequency domain analysis. Given the rather high procurement cost of systems acquired based

on their performance on benchmarks, it is striking that little attention is given to the detail and integrity of both the data and analysis performed. We propose that analysis of FTQ data take into account many features of the data.

4.8.1 Statistics for FTQ analysis

The most basic analysis is the statistical character of the data, embodied in the simple mean and variance computation over either the entire time series, or a sequence of smaller windows in order to capture nonstationarity in the data. For this discussion we will consider a sequence of N samples $S = (s_1, s_2, \cdots, s_n)$. The basic statistics can be computed as shown previously in Eq.(4.3) and (4.5). A windowed version of these statistics will provide a sequence of values. For a window of w where $N = kw$, $k \in \mathbb{N}$, we get

$$\mu_S(i) = \frac{1}{w} \sum_{j=(i-1)w}^{iw} s_j \tag{4.7}$$

$$\sigma_S^2(i) = \frac{1}{w-1} \sum_{j=(i-1)w}^{iw} (s_j - \mu_S(i))^2. \tag{4.8}$$

The basic statistics provide a global view of the data, while the windowed version provides information about the data over time. If the mean or variance increases over time, this is only apparent in the windowed version. Such nonstationary features are important to recognize in shared systems or those that have inherent interference. Reasoning about the basic statistics only causes these features to be missed.

4.8.2 Frequency domain

Often one wants to identify perturbations not only as existing, but as features with a quantifiable periodicity. To do so, moving the analysis from the time domain to the frequency domain is necessary. The Fourier transform provides the mapping from time to frequency domain. To properly enter the frequency domain, one must not only apply the Fourier transform (implemented most often as a Discrete Fourier Transform, or DFT), but quantify the sampling period and its statistics. This allows the Fourier coefficients of the various frequency coefficients to be discussed as frequencies in real time units.

Consider again a sample sequence $S(t)$ of n samples taken at periods of T_s seconds. Application of the Fourier transform carries this data into the frequency domain

$$S(\omega) \overset{\mathcal{F}}{\leftrightarrow} S(t). \tag{4.9}$$

The frequencies that can be analyzed limited to a maximum of $\frac{1}{2T_s}$Hz by the Nyquist sampling theorem. Due to the fact that the data represents a truncation of the actual sampled "signal" in the time domain, we want to remove the boundary effects that result from the start and end of the sampling period. The sudden onset and end of the signal results in the actual signal essentially being convolved with a rectangular window. Filtering using a Hamming or Hann window removes the artifacts that appear in the frequency domain due to these boundary conditions. The window is defined as

$$w[i] = \begin{cases} \alpha - (1-\alpha)\cos(2\pi i/(n-1)), & 0 \leq i < n \\ 0, & \text{otherwise} \end{cases} \tag{4.10}$$

The Hamming window is defined with $\alpha = 0.54$, while the Hann window defines $\alpha = \frac{1}{2}$. Applying the window to the data prior to the Fourier transform will remove

these boundary effects from the spectrum.

$$S_w(i) = S(i) * w(i) \ , \forall i \in [1, n] \tag{4.11}$$

Much like the time domain statistics discussed above, it is valuable to consider a short time Fourier analysis of the data to take into account potential non-stationary and transient features within the data. To accomplish this, we again consider a window w smaller than the number of samples n. For each window, we apply the Fourier transform to map the data into the frequency domain. This mapping considers a smaller number of frequency components than the Fourier transform of the entire data set, as dictated by the Nyquist sampling theorem. Furthermore, we must again apply a filter to deal with boundary effects at the endpoints of the sample sequence. The Hamming window suffices for this purpose.

The Fourier transform of the data is insufficient on its own to provide an easily interpreted picture of the behavior of the system in the frequency domain. High and medium frequency components of the signal will manifest as harmonics in the pure frequency spectrum, and make interpretation of the data difficult with respect to pinpointing specific frequency components. Fortunately, techniques from speech analysis and related disciplines can assist in this process. Harmonics show up as periodic features in the frequency spectrum itself, thus we can examine the frequency components of the data after a Fourier transform to extract this data. Formally, this technique is known as the *cepstrum*, since it is essentially turning the frequency spectrum inside out. Given the original time series data, we can define the cepstrum as

$$S_{cep}(t) = \mathcal{F}(log(|\mathcal{F}(S(t))|)). \tag{4.12}$$

Examining the cepstral coefficients (known as *quefrencies*) given the sampling

period of $S(t)$, we can pinpoint periodic features with respect to time. Obfuscation due to harmonics is removed in this case.

4.9 Conclusions and ongoing work

The work presented in this chapter is an active research area. Other groups have used similar techniques based on the FWQ benchmark to parameterize models and simulations for identifying noise [59]. That work has been highly successful and continues to provide significant insight about new and existing systems. Noticing the similarity between benchmark sampling and discrete signal processing, we have found that a carefully constructed benchmark opens up the world of DSP techniques to us for our analysis process.

In the process of this research, we were confronted by one of the performance analyst's most notorious difficulties - timer precision and accuracy. The high-precision Pentium TSC counter (and similar counters on other architectures) provides a viable solution for such simple benchmarks in identifying periodic noise that has a frequency orders of magnitude coarser than the jitter in the TSC timer. Given a high-precision timer, we were then able to create the FTQ benchmark - a tool for sampling the performance of the system in the time domain. With this measurement tool, we then began the process of exploring analysis techniques from signal processing and sampling theory to extract and identify features within the noise.

Our current work is focused on perfecting how we apply these techniques, and then interpreting the features that are extracted to correlate them with concrete hardware and software activities that are the ultimate source of noise and the resulting application performance degradation. A direct consequence of the sampling theorem is that due to the maximum sampling rate being dictated by the processor clock, there is a large portion of the spectrum at high-frequencies that is in-

accessible due to aliasing or the slight but unavoidable error on sampling interval boundaries. This contributes to the subtle but present noise that appears as low magnitude frequency components in the Fourier decomposition shown in Fig. 4.16. Application of high- and band-pass filters, filterbanks, and multi-rate analysis techniques (such as polyphase decomposition) from the DSP community may assist in separating interesting components from sampling noise. Furthermore, exploration of self-contained, computationally efficient tools to analyze microbenchmark data will assist non-experts in tuning their own cluster nodes.

Our end goal is to have the ability to measure a system, process the data and, using well-known signal processing techniques, provide quantitative assessments as to the nature of the interference, its magnitude, and its probable impact – or lack thereof – on the performance of an application on a node.

Chapter 5

Trace-driven performance sensitivity analysis

This chapter presents a simulation-based performance analysis method that is based on message-passing events recorded during the execution of distributed memory parallel codes. This research targets programs based on the MPI message-passing library, but is constructed to be general enough to incorporate other similar programming models for distributed memory parallel computing. This chapter is based on our earlier work [70] at the 2006 IEEE International Parallel and Distributed Processing Symposium.

The message passing model used to build a parallel program defines a set of process interactions that can be represented as a weighted, directed graph. Message passing events are represented as vertices. Edges between event vertices for each processor are created to represent the ordering of events, with edge weights corresponding to the duration of time spent between them. Edges are also placed between event counterparts on different processors, such as a paired send and receive, or a collective barrier. As such, the graph captures the data and synchronization flow of

the parallel program. Events can be broken into start and finish sub-events, each represented as a vertex and connected by an edge that corresponds to the time spent within the event itself. Further granularity can be used if one desires to model the transactions that are actually implemented for the various message passing primitives. Metadata related to parameters such as data sizes and types can also be included as vertex annotations or edge weights depending on the intended analysis to be performed on the graph.

Traces of parallel program executions provide the necessary data from which this graph can be reconstructed. By representing relevant communication and synchronization primitives as well defined sub-graph structures, one can rigorously define how perturbations can be simulated by manipulating the edge weights within the graph and propagating those modifications along directed edges. In this chapter, this method of trace-driven analysis is illustrated with respect to operating system interference using measurements derived from the FTQ microbenchmark in Chapter 4.

Trace-driven analysis has a long history that stretches back to batch-based mainframes [26, 65, 64], and continues through parallel profiling tools such as KOJAK [51], Vampir [75], and others. The key advantage to trace-based analysis is that it is rooted in actual, observed behavior of workloads. As such, it captures nuances and details that idealized models are incapable of. On the other hand, trace-based analysis has drawbacks. In particular, large, long running programs can produce huge traces that can be virtually impossible to store on all but the largest storage systems. This has been addressed by past researchers in projects such as Paradyn [47] and Dimemas [4], and their techniques for data reduction can be applied to this work. Similarly, trace-based analysis fails to sufficiently capture the structure of a parallel program with respect to non-determinism that may be used to increase performance. This issue is discussed in detail in Section 5.10.

5.1 Absorption measures

Simulation-based analysis works because parallel programs that do not rely solely on synchronous communications can use asynchrony to their advantage to absorb single processor perturbations without suffering significant run time changes. This is widely known in the parallel programming community, and is often cited regarding variation in communication performance and solved through overlapped communication and computation. The concept that underlies this overlap is that by initiating a communication operation before it is actually necessary and then proceeding to perform computations before its completion is required allows the communication to be delayed by a small, but valuable amount. If the amount of computation requires some time t_c to complete, then the communication operation may be delayed by an amount of time on the order of t_c without adversely affecting the performance of the parallel program.

This logic can also be applied to performance analysis of parallel programs with respect to perturbations they experience in the sequential phases of computation that are performed between communications operations. Operating system and other interference experienced by the program on computational nodes is a prime source of these perturbations. Most applications are not perfectly load balanced, leading to some degree of skew in the time that each processor reaches communications operations. This skew can have both detrimental and beneficial effects. It is detrimental to performance for one process to be perturbed, because it forces one or more other processes to spend time idle while they wait for the perturbed processes to catch up. The benefit is that these processes that are idle can experience perturbations on the order of the duration of their time spent idle without affecting the idle time induced on other processes. This allows the parallel program to experience perturbations that are absorbed by its inherent load imbalance.

Simulation of perturbations allows the degree to which a program can absorb them to be quantitatively estimated. Traversal of the message-passing graph of the program isolates the local computational phases on each processor, into which time delays are introduced and propagated over the graph based on the blocking semantics of the message-passing operations. The simulation produces two quantities for each processor representing the increase in time taken to complete the parallel program (non-absorbed delay), and the total amount of delay experienced by the program during its execution. The ratio of these values represents the *absorption ratio* that the program can experience as a function of the perturbation rate. A value of one implies that the program cannot absorb perturbations, while a value of zero implies that the program can completely absorb perturbations at the noise level simulated. This absorption ratio provides the first quantitative description of the robustness of parallel programs to local processor performance perturbations.

5.2 Trace-based delay simulation

The primary causes of performance degradation within distributed memory parallel computers are the latency of the interconnection network and perturbations to applications due to interactions with the operating system and other tasks. One technique for analyzing the performance characteristics of a distributed memory parallel program is to simulate perturbations in message latency and processor compute time, and propagate these perturbations through subsequent messages and computations to observe their effect on application runtime. This is easily modeled as a discrete event simulation, and many well defined techniques exist for building and analyzing such models [37, 31]. Unlike a general discrete event model, we chose to directly analyze the message-passing graph that results from the execution of the program on a set of nodes. The formal definitions used for delay propagation are based on

this graph. In this chapter, we introduce a performance analysis methodology that is developed to study these perturbations. This allows us to greatly simplify the model and analysis code, and provides a simple framework for defining the constraints under which the analyzer can model perturbations while still guaranteeing correctness and message order of the parallel program.

We perform and present this methodology in the context of the Message Passing Interface (MPI) library [45], but the work itself is not restricted to MPI. Parallel programs for distributed memory systems generally are implemented via primitives for passing data between processors and synchronizing computations between pairs of processors and collective processor groups. The MPI implementation of this programming model is widely used and currently very popular. Other implementations exist, such as the older Parallel Virtual Machine (PVM) [74] and the Aggregate Remote Memory Copy Interface (ARMCI) [54]. Our performance analysis methodology is applicable to all of these message-passing implementations by simply defining the primitives of the implementation in the context of the framework presented here.

5.2.1 Related work on trace-driven performance analysis

Several researchers have developed model and trace-based systems for analyzing the performance of parallel programs. Petrini et al. [59] relied on modeling the parallel program and the parallel computer before performing the analysis. This method was used to predict the performance of programs on machines prior to their construction, and to identify the causes of performance discrepancies from the predictions once the machine was constructed.

Unlike the model-based approach, other techniques are driven by traces of actual program runs. Trace driven methods have the advantage that they capture nuances in execution that arise from unique data conditions at runtime that cannot

be modeled purely by examining the static program code itself. Unfortunately, this specificity is not as flexible as model-based approaches with respect to performance prediction and extrapolation. In a trace, one loses the statistical properties of the control flow branch and join structure of the original code, limiting the potential for performance extrapolation. Furthermore, non-deterministic operations used to increase performance become fixed and deterministic in the trace.

Dimemas [4, 25], a commercial tool developed at CEBPA-Centro Europeo de Paralelismo de Barcelona, is one such tool for performance prediction of parallel programs using trace-based analysis. The user specifies the communication parameters of the target machine. A simple model [4, 25, 62] is assumed for communication which consists of (a) machine latency, (b) machine resources contention, (c) message transfer (message size/bandwidth), (d) network contention, and (e) flight time (time for message to travel over the network). Given a trace-file and the user's selection of network parameters, Dimemas simulates the parallel program's execution using the communication model. While Dimemas captures most of the parameters that affect the impact of the network on a parallel program's execution time, the model does not have similar capabilities for analyzing the operating system's interference with the application's performance.

Users who are familiar with trace analysis tools such as Vampir [75] or the Dimemas and Paraver suite will find the concept of a message-passing graph essentially identical to the visualizations that they create of parallel program execution. While Dimemas and our work are both trace-based performance analyzers for parallel programs, several key differences between Dimemas and our framework exist. 1) We seek to parameterize both the on-node noise and cross-node messaging using empirically-derived distributions from microbenchmarks. Dimemas provides an API for this purpose, but Dimemas itself does not actually perform this type of parameterization. 2) We do not require a global resolution of clocks in the trace

files required by the Vampir trace format used by Dimemas. 3) In our framework, we handle arbitrarily large trace files by streaming the trace through the simulator instead of loading it all in core. In comparison, Dimemas can handle large traces by reducing their information content in a preprocessing step. 4) We also seek to compare the effects of various architectures by using experimentally-derived parameter distributions to construct empirical distributions for deriving simulation parameters.

Dimemas provides a 'plug-in' mechanism that can be used to simulate delays both at the interconnect layer due to latency and contention, and the compute node to simulate operating system noise. As future work, we will investigate the use this plug-in mechanism to parameterize simulations using experimentally-derived empirical distributions, instead of scalar constants or idealized probability distributions. In this chapter, we extend the concept of trace-based analysis beyond the static messaging graph to a framework in which the graph is modified in a disciplined manner to model performance perturbations and their effect.

5.3 The message-passing graph concept

Consider a parallel program using a distributed memory programming model via message-passing. On a given processor, the program alternates between periods of local computation and resource usage, and interaction with remote processors via message-passing events for both data movement and control synchronization (see Fig. 5.1). If each of these periods has a time stamp at the beginning and a small amount of meta-data indicating what occurred during the period, one can easily determine what the processor was doing at any given time. We begin constructing the message-passing graph by creating a set of "straight-line" graphs (one per processor) with nodes at the beginning and end of each computation of the messaging period and an edge between successive events labeled with the duration of the period.

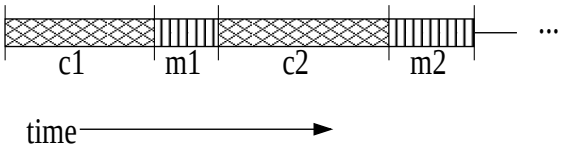

Figure 5.1: Alternating phases of computation (c_i) and messaging (m_i) over time.

Given these straight-line graphs, we now must consider message-passing activity to create the program's overall message-passing graph. The edges used to construct the straight-line graphs are referred to as *local edges* in this chapter.

During periods of message-passing activity, the processor interacts with one or more other processors depending on which message-passing primitives are invoked. Given the ordering of the events on each processor and some simple knowledge about the blocking semantics of message-passing primitives, we can easily perform a single pass over all events to decide which events on remote processors correspond to local message-passing events. Using this information, we can create edges between the coupled events on interacting processors representing the initiation and termination of a message-passing primitive. These edges that represent processor interactions via message-passing are referred to here as *message edges*.

It is vitally important for modeling consistency to create a pair of message edges for each message-passing event, although where one places the edges depends on the event being modeled. The importance of the edge pair is in recognition of the effect of local perturbations on remote nodes on the completion time of local message-passing events. The message-passing edges must capture not only latency variations between the nodes, but also allow for the propagation of remote perturbations back to all affected processors.

For modeling consistency and clarity, the model specified in this chapter embeds the semantics of the message-passing operations and their perturbations within the

graph itself. We avoid pushing the semantics of the operations to the level of the algorithm that walks the graph, as this both complicates the algorithms and makes verification and validation of the simulation more difficult. For future research, we will investigate pushing such responsibility to the algorithms instead of the graph representation for performance optimization of the analysis tool itself.

In the next section, we show how to define this graph for a subset of MPI-1 message-passing primitives [45] based on the send-receive model. Many of the remaining MPI operations share characteristics with those we describe, and our definitions can be easily extended to include them. Our methods currently do not attempt to capture the put-get semantics of other message-passing models such as ARMCI and those introduced in MPI-2 [46].

5.4 Graph primitives for a subset of MPI-1

A common method to classify message-passing primitives is to partition them into two sets based on the number of interacting processors, and partition these into two further sets based on the blocking semantics of the events. The first partition separates *pairwise* events from *collective* events. A simple send operation is pairwise, while a reduction is collective. The second partition separates *blocking* events from *nonblocking* events. The simple synchronous send operation is blocking, while an MPI `MPI_Isend` is nonblocking.

A third class of primitives exist for single node operations that are necessary, but straightforward with respect to this work. These include functions such as `MPI_Init`, which appear in the trace files and graph, but given the fact that they do not interact with other nodes, are trivial to model.

5.4.1 Pairwise primitives

The first set of primitives are the pairwise primitives. For a set of parallel processors, a pairwise event is defined as one that involves two processors exchanging a (potentially empty) data set.

Blocking

A blocking operation will not return control to the caller until it has successfully completed or encounters an error condition from which it cannot recover or proceed. The `MPI_Send` operation is a blocking primitive that sends a block of data to a receiver who posts a matching `MPI_Recv` receive operation. The MPI specification provides three forms of blocking send: the synchronous send, the buffered send, and the ready send. Each blocks until some condition has been met.

Pairwise blocking operations are easy to model in the graph, as they require a simple matching of the pair of events and the blocking nature of the operation requires a well defined begin and end relationship between the nodes. The result is that perturbations propagate through the graph to each node and preserve the pairwise relationship and event ordering.

Blocking send/receive pair

Here we present a simple graph representation of the paired send/receive operation `MPI_Send` and `MPI_Recv`.

In the presence of modeled perturbations, the end times of each operation after perturbation are determined by Eq. (5.1).

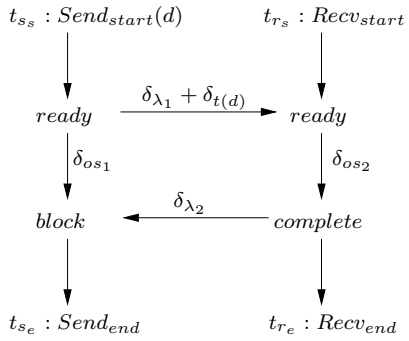

Figure 5.2: Subgraph representing a blocking send and receive pair of d bytes of data. Locations are indicated where operating system noise (δ_{os}), latency (δ_λ), and bandwidth ($\delta_{t(d)}$) are modeled.

$$
\begin{aligned}
t'_{s_e} = max(\ & t_{s_e}, \\
& t_{s_s} + \delta_{os_1} + (t_{s_e} - t_{s_s}), \\
& t_{s_s} + \delta_{\lambda_1} + \delta_{t(d)} + \delta_{os_2} + \delta_{\lambda_2} + (t_{s_e} - t_{s_s})) \\
t'_{r_e} = \quad & t_{r_s} + \delta_{os_2} + \delta_{\lambda_1} + \delta_{t(d)} + (t_{r_e} - t_{r_s})
\end{aligned}
\tag{5.1}
$$

As we can see, due to the possibility of on-node interference (δ_{os}), messaging latency (δ_λ), and perturbations that are proportional to the amount of data sent ($\delta_{t(d)}$), the completion time of send operations is dependent on the maximum of three values. These represent the original completion time (t_{s_e}), the completion time delayed by local perturbations on the sender alone ($t_{s_s} + \delta_{os1}$), or the delay due to latency in sending the message, processing it on the receiving end with potential receiver-local perturbations, and latency in acknowledging completion ($t_{s_s} + \delta_{\lambda_1} + \delta_{t(d)} + \delta_{os2} + \delta_{\lambda_2}$).

Nonblocking

It is widely recognized that significant performance gains up to some limit can be made by hiding latency to slow resources such as memory and I/O by overlapping additional computation with the resource request. As such, parallel programs often take advantage of nonblocking messaging primitives to overlap inter-processor communication with local computation to hide the high latency of the interconnection network. MPI provides primitives such as `MPI_Isend` for this purpose. These nonblocking primitives return immediately (hence the "I") to the caller, and their status can be checked at a later time. This allows the program to post data for transmission to a receiver as soon as the data is ready, and perform additional computation until the sender must block (if at all) pending the completion of the send operation.

Due to the fact that nonblocking calls immediately return, variations in latency and local perturbations on the receiving end of the transaction are not immediately apparent to the sender. We are faced with two possible situations with different consequences. First, we have a situation where the transaction is semi-synchronous. The send is nonblocking, but at a later time the sender invokes a blocking routine such as `MPI_Wait` that forces the sender to not proceed further until the communication is complete. This is easy to simulate, as it can be considered similar (not necessarily equivalent) to a synchronous send operation that has been separated into two phases. The lack of equivalence is due to the fact that multiple instances of the operation may be interleaved.

The second situation is trickier, and represents a truly asynchronous interaction between processors. In this case, the sender posts nonblocking send operations, and never blocks on the successful completion of the transaction before posting subsequent sends to the same receiver. In Fig. 5.3 we illustrate the first case of a paired send and receive followed at some later point by a pair of wait operations.

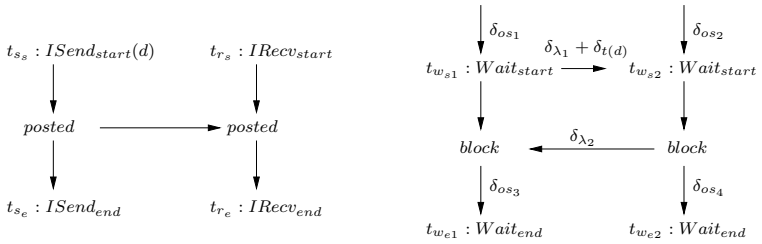

Figure 5.3: Subgraph representing a nonblocking send and receive pair of d bytes of data, and the corresponding wait operations. The send/receive pair is matched with a wait pair by matching the *status* flags that uniquely identify the send/receive transaction.

Eq. (5.2) shows the modified end times for the wait operations. Note that the end times of the send and receive operators are not modified due to their immediate return semantics.

$$\begin{aligned} t'_{w_{e1}} &= max(\quad t_{w_{e1}} + \delta_{os_1} + \delta_{\lambda_1} + \delta_{t(d)} + \delta_{\lambda_2} + \delta_{os_3}, \\ &\qquad\quad t_{w_{e1}} + \delta_{os_2} + \delta_{\lambda_2} + \delta_{os_3}) \\ t'_{w_{e2}} &= max(\quad t_{w_{e2}} + \delta_{os_2} + \delta_{os_4}, \\ &\qquad\quad t_{w_{e2}} + \delta_{os_1} + \delta_{\lambda_1}) \qquad\qquad (5.2) \end{aligned}$$

5.4.2 Collective primitives

Collective operations are used in nearly all parallel programs that require each processor to receive some amount of global state during the execution of the parallel program. These include synchronization primitives such as a *barrier*, data distribution primitives such as *broadcast*, and global application of associative operators such as a *reduction*. The presence of collective operations is often a primary source of performance degradation in a parallel program because a single slow processor will

induce idle time in all other processors. In particular, local perturbations can have a global effect on the overall program behavior.

Fortunately, modeling this is easily accomplished in the graph framework. Consider a set of p processors participating in a collective operation. Each processor has incurred some amount of simulated delay up to this point due to local perturbations and message latency. What must be decided is what the delay on each processor should be after the collective operation has occurred. A simple approach is to choose the maximum delay from the set of processors, and propagate it across all others. This is not necessarily accurate beyond a rough first approximation. The collective operation requires a sequence of network transactions to occur, and between each exists periods of local computation. This means that there is a possibility that local perturbations and network latency may cause the delay on each processor after the transaction to actually be greater than the maximum delay entering the collective.

Consider an all-reduce operation (`MPI_AllReduce`) such as a global summation. One can easily show that a butterfly messaging topology can be used to require each processor to send and receive $O(log(p))$ messages [32, 58]. This can be explicitly constructed in the graph, which allows for analysis to be performed without any special knowledge of the operation. Unfortunately, this is not space or time efficient given the fact that we know a-priori that a single collective operation can be considered equivalent to $log(p)$ periods of local computation and pairwise messaging. As such, we can simply model the collective as an edge from all p processors to a single processor, on which the $log(p)$ communication and computation perturbations are propagated, and a set of edges from this processor to all others that induces no additional perturbations, but simply communicates the maximum of this set of perturbations to every other processor.

In Fig. 5.4 we show how an AllReduce operator is modeled. In the AllReduce operation, each node must contribute local data to a global operation, the result of

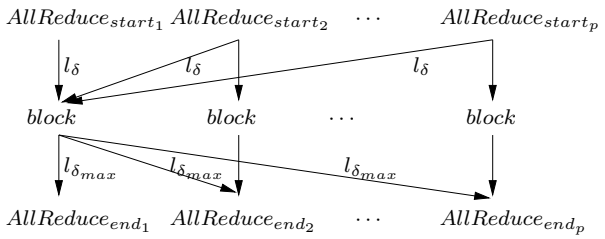

Figure 5.4: An AllReduce operator subgraph. The abbreviated noise annotations on edges are described in the text.

which is then sent to all processors. Instead of modeling the communication topology precisely, we approximate it by sampling operating system noise and latency $log(p)$ times for each processor, and labeling the edge from each i^{th} processor to the first with this value called l_δ. The maximum value of all l_δ values is computed, and propagated back out to all nodes along the return edge labeled $l_{\delta_{max}}$. This has the effect that is frequently observed in practice of forcing the slowest node (or in this case, the most perturbed node and link) to dominate the performance of the entire collective.

A simplification of this graph can be used to model a simpler Reduce operator in which only one processor holds the result after completion. In this case, three modifications are necessary. First, the message edges labeled l_δ are simplified to only sample latency once. Second, each processor has a local edge from the start node to the blocking node labeled with local operating system noise. Finally, the $l_{\delta_{max}}$ edges become unlabeled, as they are do not contribute additional perturbations themselves, but are simply required to carry the contribution of noise on the processor receiving the reduction result to those providing data to the operation.

5.5 Creation of message-passing graph

The message-passing graph that we create for analysis is generated using trace data from an execution of the program on a parallel system. Each processor creates an event trace that records the local timestamp, the event type, and event metadata for each event that occurs. This is done via the standard PMPI interface defined by the MPI specification. Each MPI primitive to be recorded is wrapped with a lightweight PMPI wrapper that records the event in a memory resident buffer. The buffer is dumped to an event trace file when it becomes full, and is then reset to empty for future events. The size of this buffer can be tuned to compensate for event frequency and overhead for I/O to dump the trace information to a file. It is unavoidable that tracing will introduce performance perturbations not present in the non-instrumented version of the parallel program. We have taken care to minimize this perturbation, but must recognize that it is present and must be kept in mind during later analysis of the program performance. For future work we will use more robust tracing tools that already exist as discussed later.

5.5.1 Avoiding clock synchronization

It is important to recognize that constructing the graph only requires pairing events across processors. The execution order on each processor makes this possible using execution ordering *only*. It is tempting, although misleading, to infer information about two processors using their local timestamps and clocks. This is related to a difficult problem in distributed systems to synchronize a set of clocks that are separated by links with non-trivial, and most importantly, unknown and statistically-defined latencies and clock drifts [15].

We take advantage of the fact that a trace of a program that ran to completion represents a message pattern that was sufficiently correct for a proper run. Each

message event is guaranteed to have a counterpart, and this counterpart can be found simply by processing each event in order on each processor. If an event is encountered and the counterpart must be found, the algorithm must simply find the next event on the counterpart processor that has not already been found that matches. This is different, and significantly simpler than deriving the messaging graph from static code, by recognizing the fact that the run occurred and the message ordering is fixed as a result. Although attempting to resolve clocks across the traces is also a possible way to align and match events, using the message ordering on each processor to regenerate the messaging pattern makes this unnecessary.

5.5.2 An implementation of the graph construction algorithm

The Chama simulation engine, described in detail in Section 5.9 implements the concepts in this chapter. It is able to process trace data and introduce and propagate simulated perturbations based on the message-passing graph in order to analyze the sensitivity of an application to message-passing latency and operating system noise. The graph is created according to the message-passing primitive semantics specified by the MPI standard and implementation specification version 1.2, some of which were illustrated in Section 5.4. The trace files are generated using a C library conforming to the PMPI standard with timestamp data provided by the high resolution, cycle-accurate timers available on all modern microprocessors.

We now describe the construction of the message-passing graph from trace data. An event is split into two subevents: a start subevent and an end subevent, which correspond to entry and exit from the message passing operation that produced the event. For finer granularities, more subevents can be added without much effort to capture implementation specific details of how the processors interact during the

message-passing primitive.

Fig. 5.5 shows a message-passing graph that our model generated from a set of trace data. For simplicity and clarity in this example, we used reduced trace data and only blocking MPI primitives. Each edge connects two subevents with an edge weight equal to the delay incurred between its source and sink subevents. The source and sink subevents need not be necessarily the start subevent and end subevent, but may be anything depending on whether the edge is a local edge or a message edge.

In order to simulate the operating system noise, the weight of a local edge connecting two subevents in the same trace is altered and the change is additively propagated through the graph to all graph nodes reachable from the sink node of the modified edge. Likewise, to simulate network latency, the weight of a message edge connecting two subevents in different traces is altered and the change is again propagated through the graph. Thus, behavior of the program under study with varying operating system circumstances and network parameters can be studied quantitatively by modifying edge weights and carrying their cumulative effect through the graph as it is traversed. This information gives a firm base on which the degree of suitability of a parallel program to a particular platform can be determined. We also can explore how varying parameters affects not only overall runtime, but regions within the graph where perturbations are absorbed or fully propagated, corresponding to tolerant or highly sensitive code, respectively.

Figure 5.5: A message-passing graph for trace data containing blocking MPI primitives.

5.5.3 Correctness

Correctness of the graph and its modification during the analysis process is vital. The process of taking traces and merging them into a single message-passing graph has the benefit of using the fact that the program did run correctly in the first place in order to create the traces. Constructing the graph based on this is simply a matter of associating events to match message end points, and this has been shown to be possible in the past as evidenced in tools such as Vampir. Correctness is important to consider though when modifying the timings of events in the process of analyzing the noise sensitivity of the program.

The key question in this process is whether the modified timings of events causes events to occur prematurely with respect to their counterparts on other processors. In a purely synchronous program, this is impossible, as the delays are propagated along the local and message edges, and all events on interacting processors are delayed in a quite straightforward manner. Nonblocking, asynchronous interactions are the complicating factor. For example, a processor that initiates a send that does not block on the successful completion of the transmission does not immediately see delays on the receiving end before it proceeds to additional events. In MPI, this is realized in the `MPI_ISend` primitive. Fortunately, in most codes that have been examined, these nonblocking calls have a corresponding blocking event that causes the sender to block on a check for the completion of the send. In essence, the nonblocking send allows the programmer to implement a synchronous send operation with the ability to inline code that does not depend on the completion of the send in between the initiation of the transfer and the check that it completed. In MPI-1, this is realized as the pairing of `MPI_ISend` with a blocking `MPI_Wait` (with `WaitAll` and `WaitSome` existing for similar blocking semantics on sets of `ISend` operations) primitive.

In the worst case, one processor issues a sequence of nonblocking sends without checking that any have completed before issuing more to the same processors. If the receiver posts blocking receives or `MPI_Wait` operations, correctness is preserved by ensuring that delays in the sends are propagated to the receiver and push the wait operations ahead to match the difference in time due to the delay. In the event that this is not possible, and both sides use only asynchronous calls with no synchronization (a possible, although questionable practice for most programs), the tool cannot guarantee that an arbitrarily perturbed graph is correct and produces a warning that this situation has been identified.

5.6 Parameterizing simulated perturbations

Given application traces, the questions that we wish to answer using the framework and tools presented here deal with how well one can expect a program to perform on a parallel computer under the influence of a set of performance influencing parameters. For example, one can execute a parallel program on a system with a minimal, lightweight kernel running on compute nodes, and then explore what amount of operating system overhead the application can tolerate before significant performance degradation occurs. The previous sections discuss the methodology for exploring the application performance under varying parameters. To best study these questions, one must also have a disciplined approach to determining how to parameterize the simulation and analysis tools.

We propose that parameters be determined using *microbenchmarks* that are carefully constructed to probe very specific performance parameters. Chapter 4 describes an example microbenchmark in great detail. Each parallel platform has a signature that is defined by the set of metrics determined by various microbenchmarks, and this signature is provided to the analysis tools, along with an application trace,

to estimate the behavior of the program on the new platform. Our current work treats parameters as random variables with a distribution parameterized by the microbenchmarks.

Two methods can be used to generate parameters for analysis given the output of microbenchmarks. First, one can estimate parameters for assumed distributions of the parameters. For example, it is generally assumed that queueing time can be modeled as an exponential distribution, and the parameter of the distribution can be estimated from experimental measurements. The second method for generating parameters is to use the data itself to build an empirical distribution. This method relies on gathering a sufficiently large number of samples such that the shape of the actual distribution is accurately captured. It is a simple exercise to show that the resulting empirical distribution approaches the actual distribution as the sample size increases, as stated by the law of large numbers [73].

5.6.1 Operating system noise

Operating system noise is the result of time lost to non-application tasks due to operating system kernel or daemons requiring compute time. A "noisy" operating system will frequently take time from applications for its own operations, while a "noiseless" operating system will allow applications to use as many cycles as possible. The effects of this noise can be quite severe, as exemplified by experiences with the ASCI Q supercomputer [59].

Microbenchmarks are available to probe systems to infer the perturbation due to operating system noise, and the data from these microbenchmarks can be used to generate empirical distributions from which our analysis tool can sample. The fixed time quantum (FTQ) microbenchmark described in [69] and Chapter 4 probes for periodic perturbations in a large number of fine grained workloads. The point-to-point

messaging microbenchmark described by Mraz [52] uses a simple message-passing program to probe the effect of noise on message-passing programs. As discussed in Section 5.4, noise is represented in the message-passing graph via edge weights on local edges. This models the additional time a processor requires to complete a fixed amount of work due to preemption for operating system tasks.

5.6.2 Interconnection network performance

The interconnection network on a parallel computer has two parameters that influence performance the most: bandwidth (how much data can be transmitted in a quantum of time), and latency (how much time is required to move a minimal quantum of data between two nodes). These parameters are easy to determine, and well known; simple benchmarks for bandwidth and latency exist for MPI and other communication protocol layers. A latency benchmark measures the variation in the time taken to send a message between two nodes. Given the lack of an accurate, high-precision global clock across communicating processors, the latency benchmark uses a traditional ping-style message exchange between two processors. A bandwidth benchmark is similar, except with messages of a significant size in one direction, with an acknowledgment returned to the sender. The size of the large message must be sufficiently large so as to make the latency component negligible in the overall time.

Two assumptions are made regarding this benchmark. First, the connection between the nodes has symmetric performance characteristics with the distribution of message latencies (from sender to receiver and vice-versa) both independent and identically distributed (*iid*). Second, two separate messages from one host to another have latency distributions that are also iid. Systems where routing adaptation and "warming up" of links occurs will violate this second assumption, and a suitable alternative tool must be employed to measure and model the appropriate statistical

distribution.

Variations in message latency and bandwidth are modeled within the graph as edge weights on message edges. Latency noise is modeled independently of the size of the message, while variations in bandwidth must be modeled as a function of the message size. Interconnect noise is also simulated using empirical distributions derived from sampled data.

5.6.3 Parameterization of perturbers

The microbenchmark data gathered from compute nodes is of the form of a time series of samples, each representing the user-process work achieved over a unit of time. This must be transformed into a form that can be used to statistically sample perturbations to induce into the simulation. One approach that is used in this work is to derive an empirical distribution based on the raw, measured data, from which we can sample. An empirical distribution is, in essence, a histogram of the data that has been normalized such that the total height of all bins adds to one in order to fulfill the requirement that a probability distribution function (PDF) has an integral of one.

The choice to use an empirical distribution derived from raw measured data versus an idealized distribution is important to consider. Many features of a parallel system can in fact be reasonably characterized using ideal probability distributions in which the benchmark is simply used to derive the distribution parameters (such as mean and variance). For example, features of the system that are best modeled as arrival counts over time (such as message arrival in a queue) can be modeled as a Poisson or compound Poisson process [37]. Other processes can be modeled as Exponential or Weibull distributions. Simple visual examination of the empirical distribution data from the FTQ benchmark in Fig. 5.6 makes it clear that the data does not necessarily

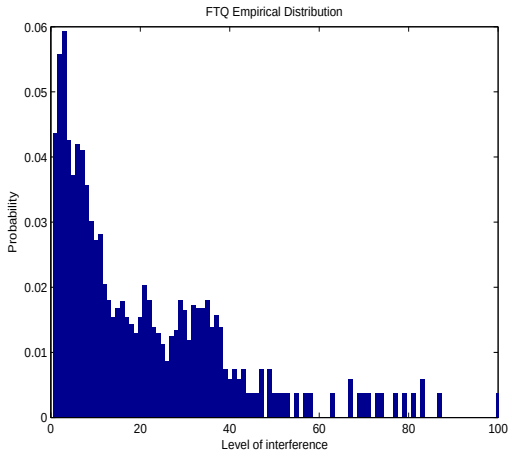

Figure 5.6: Empirical distribution derived from FTQ data. The data shows that small levels of perturbations are likely, and large-scaler perturbations are rare.

match any ideal distribution clearly. An exponential distribution can approximate it, but fails to capture the nuances that distinguish different systems from each other. The empirical distributions derived from measured data in Section 6.2 clearly show this.

Generation of empirical distribution

Suppose we start from a raw microbenchmark time series S containing n samples with a sampling period of t_s. We can construct an empirical probability distribution E by creating a histogram H_E of S with b bins, where b is chosen depending on the granularity of the source distribution of the data that we wish E to capture. Fig. 5.6 shows the data for $b = 100$. Each bin will contain the number of samples n_i that performed $[a_i, a_{i+1})$ units of work, where $a_i = \frac{i*max(S)}{b}$, for integers $i \in [0, b]$. A probability distribution requires that the total integral of the PDF is unity, so we

can define

$$\hat{H}_E(i) = \frac{n_i}{n}. \tag{5.2}$$

We want to sample from this empirical PDF. To do so, we can construct the cumulative distribution function that corresponds to the PDF of E. This is simply defined as

$$C_E(i) = \sum_{j=0}^{i} \hat{H}_E(j). \tag{5.3}$$

For a given time interval d, such as the local computational time between two message passing events, we can use this to estimate the amount of work that could occur during this period. Assuming $d >> t_s$,

$$n_d = \left\lfloor \frac{d}{t_s} \right\rfloor \tag{5.4}$$

samples of the microbenchmark capture the approximate amount of work that could have been achieved over this interval in the presence of the inherent system noise. To compute the amount of work that corresponds with this interval, we use the empirical distribution E to estimate it. First, we compute n_d samples from a uniform distribution

$$u(i) \sim U[0,1] \ , \ i \in [1, n_d]. \tag{5.5}$$

The work achieved over the time period t_d is therefore

$$w_d = \sum_{i=1}^{n_d} a_{\lfloor j \rfloor}, \text{ where } j \text{ is the point where } C_E(j) \leq u(i). \tag{5.6}$$

In the presence of higher amounts of noise, the histogram would contain less samples in bins with high work counts and more samples in bins with lower work counts. Note that in Fig. 5.6, the x-axis represents the level of noise present, which is inversely proportional to the achieved work count represented by a bin. Thus given two bins, the one with the lower noise level corresponds to the higher achieved work count. We can pick intermediate work count bins, and increase their membership while decreasing the membership of bins corresponding to higher work counts. Overall, we maintain the same number of samples, but simply simulate noise by artificially making some samples achieve less work. We can repeat the above process with this new sample set $\hat{S}$. The mean work performed in a sample given this new distribution decreases from the original. Therefore the number of samples $\hat{n}_d$ required to achieve the same amount of work w_d increases.

We now use the new distribution $\hat{S}$ to create a new CDF $C_{\hat{E}}$ from which we sample using Eq. (5.6). In this case, instead of sampling to determine the amount of work achieved over a period of time d, we sample as many times as necessary such that their total work is approximately w_d. Due to the modification of the empirical distribution to model higher noise, if we use the set of uniform samples $u(i)$ to compute the new amount of work performed over n_d samples, $\hat{w}_d \leq w_d$. Therefore we need to use more samples from U to make $\hat{w}_d = w_d$. Assume that we require $\hat{n}_d > n_d$ samples to achieve the same amount of work as in the unperturbed case. This means that in the presence of simulated noise, the time period d increases from $t_s n_d$ to $t_s \hat{n}_d$.

Using this, we can simulate perturbations as causing an event that required $t_s n_d$ time units to now take $t_s \hat{n}_d$ time units, offsetting the event by $t_s(\hat{n}_d - n_d)$ time units. Any microbenchmark data that results in a distribution of values, ranging from operating system perturbations to message transmission latency, can be used to model perturbations in this manner. In a real system, nearly all values measured

by microbenchmarks have a stochastic nature and are best considered to represent empirical probability distributions of the actual value being measured. Thus this method for parameterizing perturbations can be applied to capture the stochastic nature of a general set of sources in order to drive the sampling used for simulations.

5.7 Implementation and example application

The initial implementation of the tools for analyzing traces includes a simple PMPI-based tracing generation library and an analyzer that inputs these traces, constructs the message-passing graph, and allows for a very simple parameterization of edge weight modifications to explore noise and latency variations. The analysis tool uses the algorithm described in Section 5.5 to connect individual traces for each processor with message edges. To avoid the obvious limitations imposed by memory constraints, the analysis tool uses a windowed approach to building the graph. This is particularly important to consider given the number of events in a long running, high processor count job.

Given the set of performance parameters related to noise in the operating system on processors and the interconnection network connecting them, the analysis tool processes the graph in the following manner. As the graph is created using subgraphs as described in Section 5.4 the δ values that are indicated as edge weights are generated by sampling the distributions associated with the parameters. The original message-passing trace has edge weights on local edges corresponding to the time intervals observed in the run that generated the trace. Message edges are weighted zero originally, as the effects of latency and bandwidth are already embedded in the timings of the actual events that occurred. Simulating additional delays in messaging is achieved by marking message edges with positive values. As the graph is streamed through the tool, the $max()$ operators defined in Section 5.4 are applied to modify

the times of each node in the graph based on the simulated perturbation deltas added to both message and local edges. The end result is a final modified timestamp on the final node for each processor corresponding to the `MPI_Finalize` event.

From this new completion time, we can observe how running times for the overall program and individual processors increase in the presence of varying degrees of noise. For example, if we generate a trace on a system with relatively low noise (such as a bproc cluster as discussed in [69]), we can parameterize the simulation with performance parameters measured on a system with higher noise to explore how the program can be expected to perform on a system composed of higher noise processors.

We do not currently explore the possibility of determining how a trace taken on a high noise system would run on a system with lower noise. A similar methodology could be applied by introducing negative edge weights, but this sort of analysis is being left for future work.

5.7.1 Token ring

A token ring is one of the simplest messaging topologies found in realistic parallel programs. In n-body simulations, it is occasionally true that the n^2 particle interactions must be computed directly instead of using approximation algorithms that require $O(n \log n)$ or $O(n)$ computations. For p processors, it is possible then to divide up the n particles into sets of $\frac{n}{p}$ on each processor. Each processor p_i then packages up the set of particles that it "owns", and passes it to the $(i + 1 \ mod \ p)^{\text{th}}$ processor. This processor computes the interactions between its local particles and those contained in this "token" containing a particle set from some other processor. This set is then passed on to the next processor as before, and this is repeated p times until each processor receives the token containing its local particle set, at which time

each processor has computed the influence of all n particles on their local set.

Our initial experiments verify the intuitive behavior that one would expect from a fully synchronous program as this. We performed a traced run on 128 processors of a ring-based program, and varied the degree of perturbations from none to a mean of 700 cycles worth of perturbation at 100 cycle increments. The resulting change in running times increases for each processor that matches the 100 cycle increments multiplied by the number of traversals of the ring. For example, if the ring was traversed 10 times with each processor injecting 100 cycles of noise for each message, the runtime of each processor increased by approximately 10*100*128 cycles.

A study of the simulation used for five different real MPI codes appears in Chapter 6.

5.8 Future work for Chama

The tools we designed and implemented are developed to explore the feasibility and algorithmic aspects of this method of performance exploration. Two major areas of work are in need of immediate attention. First, we plan to use existing tracing libraries that provide a more complete treatment of the MPI specification, in addition to allowing traces to be generated for other message-passing and shared memory parallel programming tools. The library we are exploring, KOJAK [51], provides the EPILOG tracing format and accessor library. The second area of work is to provide a mechanism to provide a richer set of parameters to the simulation, and maintain a history of analysis experiments that are performed using our tools. We would also like to investigate modeling reduced noise from that observed in the traced runs to explore how performance could be expected to change if the run was performed on a system with *less* noise.

We have presented an analysis methodology and prototype of a performance analysis tool driven by message-passing traces, which is scalable and ensures correctness of the analysis that preserves message ordering true to the trace-generating run. We discussed how operating system and interconnect parameters can be generated and integrated into our analysis methodology. We model the application as a message-passing graph, which is traversed in the same order as the execution order of the original parallel program. Enforcing no changes in the order of execution ensures correctness of the model in the presence of blocking and nonblocking message-passing primitives. Our windowed graph generation technique allows us to analyze traces of arbitrarily large size on systems with limited memory, thus making it fully scalable. Since trace-based simulation reflects application behavior on real machines under internal data states for real runs, the results are expected to be more accurate for a given processor count than an idealized model at the cost of restricting extrapolation abilities.

While the tools are still early in development, currently supporting only a subset of blocking, nonblocking and collective MPI primitives, this work introduces a promising methodology for analyzing parallel program performance taking into account their *actual* runtime behavior for real problems. In the future, we also plan to expand this performance analysis to support more of the MPI-1 primitives, in addition to other parallel programming paradigms including but not limited to extensions present in MPI-2 and other distributed memory models such as ARMCI. These primitives represent what are known as *one-sided* communications operations.

5.9 Chama simulation engine structure

The simulator development went through two phases. The first implementation was a Standard ML [48] tool for processing traces and reconstructing the message

passing graph. Standard ML was chosen for its richly expressive type system, in which most of the details regarding data structure representation were handled by the language itself. Other languages such as C and Java pollute the semantics of the graph reconstruction in the code with details related to memory management and data structure maintenance. Its goals were two-fold. First, it was designed to make it simple to simulate operating system noise and other non-communication related effects. The second goal of the prototype was to investigate how a trace-driven simulation could be designed to minimize its specificity to any single message passing model, and to eliminate the need for global time synchronization. This prototype proved to be functional and met its goals for a small subset of the MPI-1 standard, but was poorly engineered with respect to extensibility. Furthermore, although functional languages can achieve high performance, doing so is difficult and can make code difficult to read and maintain. Finally, the lack of community acceptance of languages not commonly used would hinder the adoption of the tool by new users.

After writing and experimenting with the initial version of the tool, a second version was written using lessons-learned from the original. This new version has proven to be easier to maintain and extend, both in terms of the communication semantics that are being modeled and the manner by which simulated perturbations are introduced and then tracked. It implements most primitives required for collective and two-sided message passing, the choice of which was driven by the needs of a set of MPI-based applications of interest. The set of MPI operations supported both by the tracing library and the simulator are shown in Table 5.1. Purely local operations such as `MPI_Abort` and the `MPI_Type` family are not necessary to simulate, and are implicitly supported.

Chama is a single threaded, event driven discrete event simulator that uses the message-passing graph formalism to model noise propagation. It processes all events

MPI_Allgather	MPI_Allgatherv	MPI_Allreduce
MPI_Alltoall	MPI_Barrier	MPI_Bcast
MPI_Bsend	MPI_Comm_create	MPI_Comm_dup
MPI_Comm_free	MPI_Comm_group	MPI_Comm_rank
MPI_Comm_size	MPI_Comm_split	MPI_Finalize
MPI_Gather	MPI_Gatherv	MPI_Group_incl
MPI_Group_intersection	MPI_Group_range_incl	MPI_Ibsend
MPI_Init	MPI_Iprobe	MPI_Irecv
MPI_Isend	MPI_Issend	MPI_Probe
MPI_Recv	MPI_Reduce	MPI_Scatter
MPI_Scatterv	MPI_Send	MPI_Sendrecv
MPI_Ssend	MPI_Wait	MPI_Waitall

Table 5.1: MPI message passing primitives supported by the simulator and tracing library.

in the order in which they occurred, maintaining the cross-process relationships that occurred and manifested as blocking in the original programs that were traced. These relationships correspond to the cross-process edges in the message passing graph, not only to represent the blocking relationship of events that occurred, but to capture the path by which perturbations would propagate at simulation time.

The simulator is based on decomposing a parallel program into a set of entities that represent the relevant local state on each process, and the global state that groups of processes share.

- **Process**: A process has a local clock that advances as events complete and simulated perturbations are introduced. It also contains a set of messages that it has issued either synchronously or asynchronously to other processes, and a set of messages that it is expecting from other processes. It also may be in a blocked state that requires another one or more processes to advance their state in such a way to allow the process to proceed. A process also has a set of contexts in which it coexists with other processes, manifested in this

implementation as MPI communicators and process groups.

- **Communicators**: The notion of a communicator is borrowed from MPI, although in the simulator it can be used to represent similar logical constructs found in other parallel programming models. A communicator is simply a set of processes that can participate in collective operations and exchange pairwise messages. Often a parallel programmer will decompose a large set of processes into smaller subsets that will perform work as collective entities. This component of the simulator maintains the state necessary to capture the behavior of these groups in such a way that does not require all processes in the entire parallel program to exist in a single global group.

- **Groups**: The representation of process groups is separated from the communicator that collective events occur within. This allows the set-theoretic operations used to create process groups to be separated from the structure that implements the communication operations. This separation of groups from communicators is inspired by the same separation present in MPI, but is applicable in other message-passing models that also employ logical process groupings. The communicator and group structures also are used to provide a rank mapping for a process from the logical processes groups to the physical process itself.

- **Communicator Contexts**: A communicator context is used by a process as a translator between the local process view of a process group and the global set of processes. Each process within a group has an independent namespace for identifying process groups, and operations on the global set require a translation from the local to global namespace. This is due to the fact that a subset of processes may collectively create a new process group G_A not seen by other processes. If a different subset of processes create another process group G_B that includes processes also involved in G_A, then the identifier for this group

on $G_B \setminus G_A$ will differ from that on the processes in $G_B \setminus {G_A}^c$.

- **Events**: The simulator is driven by events that were generated by runs of real parallel programs. An event represents a phase of interaction with other processes during runtime. The time between events is time spent performing purely local work by a process. For example, a process may generate a receive event followed by a send event. Between these events, the process may be computing based on the data received to generate the new data that is then sent later. Events have time and data related information, and serve as the mechanism by which processes and communicators change state. They also contain event-specific metadata describing the interactions with other processes that the event represents.

- **Perturbers**: In its basic form, the simulator simply executes a sequence of events from parallel event streams in the order that they occurred. The goal of the simulator is to use this stream in conjunction with a parameterized perturbation injector that artificially induces delays in event start times or completion times based on the performance parameters to study. Perturbers within the simulation are present during event processing on processes and communicators, and modify the occurrence times and durations of events.

Other structures exist within the simulator for tasks such as trace I/O, statistical bookkeeping, and sampling of random variables of various distributions. The perturbers are used to induce simulated perturbations in timings for modeling parameters such as operating system interference.

Tracing overhead

In any profiling or tracing study, without external hardware that monitors the memory bus or interconnection network, there is some level of perturbation induced by

the measurement software itself. This is due to resource sharing of the measurement tool with that which is being measured. The data gathered during the execution of an instrumented program only approximates the behavior of the program on its own. Care was taken in constructing the PMPI-based tracing library to minimize this perturbation, specifically in terms of memory consumption and trace I/O overhead. Each MPI call made by the user program was first passed through a tracing library wrapper, which logged all necessary data about the event, before being handed off to the MPI implementation. The MPI standard specifies a well defined interface for such wrappers, known as the PMPI interface. Each PMPI wrapper had a form similar to that shown in Fig. 5.7 for the `MPI_Send` subroutine.

The "LOG" macros index into a trace buffer to store specific information into the structure representing the current trace event. This buffer is composed of 8K event entries, each of which requiring 108 bytes of storage. The resulting trace buffer size on a 32-bit Pentium 4 based system is approximately 885 kilobytes. The final trace buffer advance call simply increments the position in this trace buffer. If the number of events exceeds the buffer size, the data is written to a storage device and the current event pointer in the buffer is reset. Tuning the buffer size has two tradeoffs. By increasing the size of the buffer, the frequency of highly-perturbing I/O operations is reduced. On the other hand, increasing this buffer size consumes memory that could be otherwise used by the application, and introduces potential cache effects that may degrade application performance.

Future versions of this tool will use more mature tracing libraries. The initial implementation chose to use a custom trace format to avoid the need for global clock synchronization in the trace post-processing and to use a trace format that was most efficient for implementing with the simulator design. A text-based format is significantly larger than necessary, but has the benefit of easy readability for debugging during the development process. Moving to existing trace formats will not only al-

```
int MPI_Send(void *buffer, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm) {
  int retval;

  LOG_ETYPE(EVENT_SEND);

  LOG_COMM(comm);
  LOG_ORANK(dest);
  LOG_SIZE(count);
  LOG_TAG(tag);
  LOG_DTYPE(datatype);
  LOG_STIME();

  retval = PMPI_Send(buffer, count, datatype, dest, tag, comm);

  LOG_FTIME();

  advance_tracebuffer();

  return retval;
}
```

Figure 5.7: A PMPI implementation of the `MPI_Send` wrapper function.

low traces to be smaller, but will allow analysis on a single application trace with both the Chama tool described in this thesis and other tools used in the parallel computing community.

5.9.1 Message passing graph construction

The message passing graph is constructed in a memory efficient, single pass fashion with delay injection and propagation occurring as it is traversed. The memory requirements to simply reconstruct and walk the graph are $O(p)$ for a p processor trace.

Additional memory is required by optional statistical trackers that record per-event or per-event-class (such as `MPI_Allreduce` collectives) data. The goal of low-memory requirements is to facilitate simulation-based studies of very large processor count programs for massively parallel systems such as the IBM BG/L machine or large scale ASCI clusters.

Consider a p processor program containing n_c process groups and communicators at runtime. One instance of the `Process` object is created for each processor, and one instance of the `Communicator` and `Group` object is created for each communicator and process group. For each process, a `CommunicatorContext` object is created for all communicators known to the process either through membership or communicator constructor operations. This object provides the required mapping from the local process namespace for communicator IDs to the global communicators that logically bind process groups together. Each `CommunicatorContext` maintains five indexed sets for bookkeeping to reconstruct pairwise messaging events. They are as follows.

1. A set of $p-1$ "mailboxes", corresponding to sets of pairwise message events from each other process that have not yet been matched with a local event.

2. A set of $p-1$ sets of "in-flight" messages corresponding to pairwise messages originating from the local process that have not yet been consumed by their matched events on remote processors.

3. A set of outstanding requests, corresponding to all requests (`MPI_Request` instances) that have been posted by asynchronous messaging calls, but not yet fulfilled by the remote process.

4. A similar set of requests indexed by remote process index.

5. A set of fulfilled requests. Each fulfilled request is a pair containing the original request that was posted, and the remote event that fulfilled it.

Each of these structures can potentially require $O(p^2)$ space, which would be prohibitive for large scale process counts. This would occur if each process initiated a large number of asynchronous messages to all other processes. Fortunately, this is a highly unlikely situation in practice. The reason for this is that operations with this sort of logical all-processors message pattern are implemented efficiently using collectives. Users generally do not, and should not, implement these operations themselves.

Collectives are expressed over process groups in MPI via the `MPI_Comm` structure. They include operations such as all-to-all data exchanges, implementation of reduction trees, data broadcasts, and synchronization barriers. Collective events are either *rooted* (such as an `MPI_Bcast` or `MPI_Reduce`) or not (such as an `MPI_Allreduce`). The `Communicator` object contains data structures for implementing collective operations, each requiring $O(p)$ space.

1. A vector of `Process` instances that have not yet reached a collective that is pending. Those processes that are not in this list have reached the collective and are blocked from proceeding.

2. A mapping of `Process` instances to the `CollectiveEvent` object that they have reached.

3. A set of `Process` instances that have reached a rooted collective before the root, and are blocked until the root process reaches the collective.

The simulation reads events from processes in a round-robin fashion, skipping those that are blocked pending completion of events on other processes. No global clock synchronization within the traces is required, as the blocking semantics defined by the MPI standard alone are sufficient to reconstruct the message passing graph given the communicator and process state described above. The only modification to

the trace files that is required is the resolution of non-deterministic operations into the deterministic decisions that were realized at runtime. These correspond to the `MPI_ANY_TAG` and `MPI_ANY_SOURCE` options available for pairwise receive operations. Issues due to these parameters are discussed in detail in Section 5.10.

5.10 Limitations

Trace-based simulation and analysis has both strengths and weaknesses. It is clearly desirable over idealized models in some cases because it gives a representation of true program behavior under the dynamic data conditions that occur during execution. These data conditions result in unique control flow and resource consumption patterns that are difficult to distill into a simple, generalized model of the program. On the other hand, parallel programs can take advantage of some level of non-determinism to increase performance. Unfortunately, this non-determinism limits what kind of result can be derived from traces of such programs.

There are two primary limitations of trace-based tools that are discussed here. First, they are not appropriate for scalability studies. Second, they can provide only limited information about the performance sensitivity of programs that rely on non-deterministic control flow. These limitations are perfectly acceptable, as there exist other tools and analysis methods that are better suited to answering questions that trace-based tools cannot. No tool or method can answer every question, so a performance analyst must be prepared to apply a suite such that each question of interest is answered using the appropriate method.

5.10.1 Scalability studies

The most common question encountered during the lifetime of this work was how well it can predict the runtime of a program that was executed and traced with p processors on $2p$ or $10p$ processes. The short answer is that it cannot, and never will, because it is not a tool for that purpose. It is related though. Given a tool for analyzing scalability, one can determine a set of predicted runtimes for a parallel program on a sequence of processor counts. The tool described here allows one to ask, for each of those processor counts, to what degree the runtime will vary given a trace of the program execution on the given processor count. A sequence of experiments that are used to extrapolate scalability can also be used to extrapolate a *sensitivity envelope*, providing a disciplined estimate of the uncertainty in the predicted, scaled runtimes.

The reason trace-based methods are not appropriate for scalability studies of all but the most simple parallel programs is that the messaging pattern that occurs within a parallel program is very often a direct function of the processor count. Furthermore, given a messaging pattern, the amount of work and manner by which data flows between processors can also be a function of processor count.

It is not impossible though to use traces to motivate scalability studies. For example, given a trace of a p processor run, it is feasible for the $2p$ case to essentially be composed of the p processor trace data duplicated, with a small number of messages introduced at regular intervals representing the communication between the two processor sets. If the algorithm is using a tree-based messaging topology, the messaging pattern for the $2p$ case may be inferred (most likely with some level of human intervention) from the p and $p/2$ trace data. The $p/2$ traces may be derived from the p trace without requiring an additional tracing run at the lower processor count.

Others have explored the possibility of trace-driven scalability studies, including the Extrap project from the University of Oregon [63, 50].

5.10.2 Non-determinism

It is quite possible, and occasionally desirable from a performance perspective, to allow some amount of non-deterministic execution to occur within a parallel program. The basis of this is that one process in a parallel program will occasionally reach a state where it requires data to continue, and this data can be provided by one or more of its parallel peers. Many algorithms specify precisely which processor must provide this data due to data distribution and dependency structure. The successive over-relaxation algorithm is a simple example of this. In some cases though, there is no explicit single processor that can provide data. If a set of processors can provide equivalently useful data sets to the processor that requires data, then choosing the first to arrive will reduce the idle time spent waiting to proceed. Similarly, if a processor must process data from each of a set of processors without requiring any ordering of the data sets, then it would be wise to iteratively take data as it becomes available, in the order that it becomes available.

The problem with this sort of non-deterministic messaging pattern is that each run may produce different traces as data conditions within the program change, or interference from outside the program affects performance on each processor. A processor may have a variety of control flow patterns that depend on which process it received data from, or similarly, whether or not it was the process from which the non-deterministic receive was fulfilled. Due to this potential change in control flow based on which message source was used at runtime to fulfill the non-deterministic receive, the messaging pattern that follows it may also change.

A trace only can truly represent one instance of the program behavior in which

each non-deterministic operation was executed with precisely one of the possible sets of parameters. Each non-deterministic choice was collapsed into a single deterministic choice at runtime.

Implementation details

Message passing operations of this form are provided by the MPI standard via the `MPI_ANY_SOURCE` and `MPI_ANY_TAG` constants that can be used as the source or message tag parameters to the `MPI_Recv` and `MPI_Irecv` operations. The program is able to determine which source fulfilled the receive operation at runtime by examining the contents of the `MPI_Status` after the blocking receive or wait operation (in the case of `Irecv`) completes.

The trace file must allow post-mortem analysis tools to determine which message source was used to fulfill the receive operations. In the case of a blocking receive, this is easy, as the status field that is populated upon completion of the call can be immediately read and traced. Non-blocking receives pose a small complication. Consider the following code sequence in Fig. 5.8.

The peer process used to fulfill the non-blocking receive operation is not known until the blocking wait executes to stop execution until the pending receive completes. Only after the wait returns can the process determine what source was used to fulfill the receive. At runtime, the tracing library should not attempt to backtrack and make sure that the receive event is annotated with the precise source and tag that fulfilled it. This would require far too much memory and computational impact on the user code, inducing more perturbations than necessary into the run and causing the trace to be even less representative of the code itself. It is quite simple to execute a post-mortem processing step (that requires one pass only over the trace files) that resolves the non-deterministic receive events with the deterministic decisions that

```
MPI_Request req;

MPI_Irecv(buf,MPI_ANY_SOURCE,MPI_ANY_TAG,&req);

/*
 * code unrelated to the Irecv operation
 * ...
 */

MPI_Wait(&req);
```

Figure 5.8: An illustration of non-determinism that can be implemented using pairwise receives.

were made at runtime. A small tool to perform this post-processing is provided with the PMPI tracing library used in this work.

Chapter 6

Experimental studies

To revisit the hypothesis underlying this work, the goal is to define a methodology for rigorously measuring relevant performance characteristics of a parallel machine, and use these measurements to drive analysis of application codes. The previous chapters outline both an example performance measurement technique for inferring operating system interference as experienced by applications, and a trace-driven simulation technique for examining the sensitivity of applications to runtime perturbations.

In this chapter the techniques described in earlier chapters are applied to study applications and their performance sensitivity to performance parameters such as network performance and operating system noise. Results are also given showing the use of the FTQ microbenchmark. Data is also shown with respect to the cache effects seen by FTQ that relate to self-interference within the microbenchmark data. The applications are chosen to represent real workloads found on real parallel systems.

The applications of interest are taken from multiple sources to represent workloads of importance to different scientific computing communities. Sweep3d [36] is an open benchmark that represents a typical kernel found in many scientific applications. Three codes from the NAS parallel benchmark suite [5] are chosen to

represent simple kernels that exhibit a wider variety of structure than the simple Sweep3d program. The ASCI FLASH [23] code is an open, university developed code that represents a full application that targets large scale ASCI computing platforms. These codes are also chosen due to their open availability without export restrictions to maximize potential for other researchers to reproduce or extend the results presented here.

6.1 Simulation

The experimentation presented in this chapter is performed using a trace driven simulator based on the work presented in the previous chapter. The parameter that is focused on for modeling and simulation is operating system interference. Before demonstrating the functionality of the simulator, we will examine the performance of the simulator itself. This is important from a user perspective because not only must the tool provide a result that one can believe, but it must do so in a reasonable amount of time.

6.1.1 Simulator performance

We test the Chama simulator for its own performance using traces for various processor counts and application sources. It must be able to process a single trace and produce output in a reasonable amount of time, preferably on the order of minutes for a single combination of perturbation parameters. We profile Chama using the standard Java profiling interface accessed from the NetBeans IDE. The simulator was not optimized in any way, and relied solely on data structures and algorithms provided by the Java standard library.

The runtime of the simulator for the largest trace data, a 1.6 million event trace

of a 32 processor run of the ASCI FLASH code, was 3.5 minutes on a 1.5GHz Pentium M laptop with 1.5 GBytes of main memory. An identical run performed on a 2GHz AMD Sempron 3300+ based workstation with 512 MBytes of main memory completed in approximately the same amount of time. Other traces of shorter length from the other codes examined took less time. The runtime for a simulation run is most dependent on the number of events being simulated, and less so on the sort of events being simulated.

The event types that are simulated have varying complexity of implementation. Basic point to point messages require $O(1)$ time to handle, with $O(n)$ time to identify counterparts on remote processors where n is the number of messages that have been posted and not fulfilled at any given time. The value of n is generally small, and only increases if a large number of asynchronous messages are posted between a pair of processors before their counterparts are reached. In the FLASH code, n was between 1 and 5.

Collective events require lookups in data structures to determine the state of the p processes participating in a single operation. The Chama tool chooses to spend memory in the interest of reduced time complexity in this case. By using tables indexed by the processor rank, lookups require $O(p)$ space and take $O(1)$ time. The blocking semantics of collectives means that only one collective event is pending at any given time, although cases can be created where multiple rooted collectives can be in progress at once. This was not observed in any of the traces that were used for development and testing, and may even be prohibited by the underlying MPI implementation.

The results of profiling on all of the test cases showed that the primary performance bottleneck within the simulator was in the trace I/O interface. Most of the time during execution was spent in one subroutine that reads single values off of the I/O stream to populate the fields of the `SimEvent` object used by the simulator.

The trace format employed by Chama at the current time is very space inefficient, and requires ASCII text to be read and converted into the corresponding integer or double precision value. This penalty was paid in the interest of human readability during development, and can be easily remedied by adopting a more space efficient binary format that requires less parsing and type conversion at runtime.

Finally, in performing a study of the performance change as perturbation parameters are varied, the simulator was run in a batch mode on a single workstation. For each noise level (corresponding to the probability of a perturbation in a period of simulated time), a small (5-10) number of runs were performed to ensure that the results were not larger or smaller than expected due to specific choices of pseudo-random number generator seed. For 20 noise levels with 5 runs each, this corresponded to 100 runs of the code. On this single workstation, this required approximately 6 hours of computational time. Each run is completely independent of each other, so this time can be drastically reduced by utilizing multiple workstations working in parallel. A laptop and two workstations could easily perform a similar study in part of an afternoon with current capability systems and the unoptimized Chama code. Optimization within Chama will easily reduce this time significantly. The times required for each set of simulation runs used to generate the data presented later in this chapter are shown in Table 6.1.

6.2 FTQ results

This section presents results for the FTQ microbenchmark in two areas. First, we will present the results of profiling FTQ for hardware counter data in order to illustrate the cache effects that FTQ experiences that manifest as self-interference in the FTQ output data. Second, we will show the output of FTQ under different interference levels to illustrate how the interference is revealed. We also demonstrate the

Trace input	Noise levels	Runs/level	Time (min.)
24 CPU Sweep3D	90	20	26
24 CPU Sweep3D (1)	45	10	6
24 CPU FLASH	45	20	497
24 CPU FLASH (1)	45	5	125
16 CPU NAS LU	45	20	252
16 CPU NAS LU (1)	45	5	63
16 CPU NAS CG	90	20	91
16 CPU NAS CG (1)	45	5	11
16 CPU NAS MG	90	20	39
16 CPU NAS MG (1)	45	5	5

Table 6.1: Simulation run times for different parameter studies. Rows labeled (1) indicate simulations of single process noise.

conversion of measured FTQ data into empirical distributions suitable for sampling.

6.2.1 Cache effects and self-interference

The experiments in this section are performed on an AMD Sempron 3300+ based workstation. This configuration features a 64Kb L1 instruction cache, a 64Kb L1 data cache, and a 128Kb L2 cache. Each cache uses 64 byte lines. The measurements are taken using the PAPI version 3.2.1 hardware counter accessor library using a version 2.6.11 Linux kernel patched with the PAPI provided 2.6.x version of the `perfctrs` library. The FTQ code is instrumented using the TAU profiling library version 2.15.2, with the PAPI counters for data and instruction misses for both the L1 and L2 caches.

Instrumentation is inserted to capture data within three portions of the FTQ code. The main region, `core`, encapsulates the entire main loop that included the sampling loop, `loop`, and the two array operations used to save both the sample work count and the time elapsed for the sample (`data`). In the code shown in Fig. 4.4, the

	Core	**Loop**	**Data**
L1 ICM	6013	1515	1502
L1 DCM	127	47	**1308**
L2 ICM	59	9	2
L2 DCM	58	3	5

Table 6.2: Cache miss statistics for a sampling interval of 2^{20} cycles over 5000 samples

	Core	**Loop**	**Data**
L1 ICM	12030	3038	3004
L1 DCM	201	107	**2369**
L2 ICM	142	33	10
L2 DCM	60	5	8

Table 6.3: Cache miss statistics for a sampling interval of 2^{20} cycles over 10000 samples

`core` region spans lines 3 through 17. The `loop` section spans lines 7 through 10, and the `data` section spans lines 12 and 13. The profile data represents exclusive counts. This means that the `core` data includes counts that are not contained within the `loop` or `data` regions within the main region. Tables 6.2 and 6.3 show the results of the profile for two different parameterizations of FTQ.

Given that we wish to use FTQ to probe for operating system perturbations that occur at time scales significantly longer than processor cycles, FTQ must execute a very large number of samples to reveal them. The Nyquist sampling theorem defines the number of samples required at a specified sampling rate to allow for reconstruction of periodic features of the signal from sampled data. Thus the size of the sample data storage will be very large, and will easily exceed the cache line size of both the L1 and L2 caches in even the most capable modern systems.

Fortunately, FTQ fills the sample data storage in a regular linear pattern, which

means that cache misses will occur at highly regular and predictable intervals given a specific hardware configuration. As we can see from the data, the cache miss rates indicate a high correlation between the number of samples taken and the precise region of code. In the first case with 5000 samples, we observe 1308 L1 data cache misses, corresponding to a cache miss over 26.16% of the samples. This is approximately one quarter of the time. This makes perfect sense, due to the 64 byte cache lines on the L1 cache. If each iteration requires two double precision (8 bytes each) values to be stored, we would expect four iterations to fill a cache line, since this corresponds to eight double precision values, or 64 bytes. The first iteration from each of these blocks of four would induce the miss to load the cache line, hence an expected miss rate of approximately 25%. Similarly, for the larger iteration count of 10000 samples, this trend continues with 23.69% of the samples observed to induce an L1 data cache miss.

6.2.2 FTQ data

In this section, data are presented from FTQ to illustrate how it is able to clearly differentiate varying levels of noise within benchmarked systems. Data are presented from two perspectives. First, data are shown under different noise levels to illustrate the change in the raw FTQ output that provides quantitative evidence of both the existence and magnitude of noise. Second, data are shown for a fixed noise level with varying granularity work quanta within FTQ. This illustrates the importance of tuning the work quantum granularity for platforms being measured. These results were presented at the SIAM Parallel Processing for Scientific Computing conference in February, 2006.

FTQ under varying noise levels

The data shown in Fig. 6.1 demonstrate the data produced under varying noise levels on an IBM X41 laptop. The use of a workstation platform to illustrate noise allows a wide spectrum of noise levels to be explored. Three different cases are shown. The first is measured before the operating system has started any daemons or other processes, and shows the lowest noise level. This is a baseline measurement as described in Section 4.6. All samples achieve approximately equal amounts of work within the microbenchmark, and perturbations are infrequent. The second case is measured after daemons have started and the system is in a state where user-space processes can execute. Perturbations are more frequent, and the variance in the work achieved per sample increases. Finally, the last case illustrates the worst-case scenario, where not only operating system tasks are executing, but processes related to the GNOME desktop environment are also running and competing for resources. The noise is very high, and the variance seen in FTQ work achieved in each sample is very large.

Each plot of the raw data is shown on equal axes to illustrate the relative differences in the work achieved per sample. Figure 6.1 also shows the cepstrum corresponding with each FTQ time series for the first 3000 quefrencies. No additional processing has been performed on the data other than applying a basic Hamming filter to remove boundary effects in the spectrum due to finite sample lengths before applying the Fourier transform. As we can see, the number of peaks above the noise floor increases as the level of interference in the system is increased. The pre-`init` case shown in the top two plots shows only two significant spikes in the cepstrum. The second row of plots shows that the number of peaks increases, as does the amplitude of their values.

The data shows some degree of nonstationarity, as the raw data (on the left)

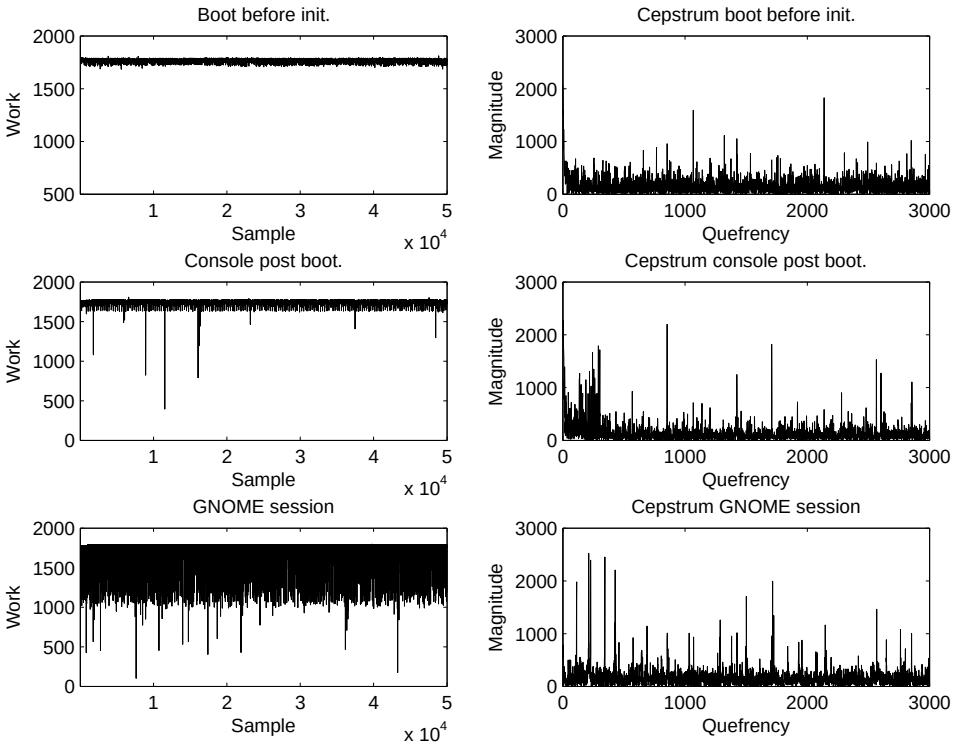

Figure 6.1: Three FTQ time series taken on an X41 laptop under varying noise conditions.

has more perturbations at early time than late. This early time difference in the data is also apparent in the final row of data, and can be attributed to operating system scheduler adaptation as the process executes over time. The final data set shows a larger number of peaks than the previous cases, with peaks reaching a higher amplitude.

This experiment shows that FTQ data clearly differentiates between noise levels, both in the raw data and the cepstral representation. In Section 5.6.3, it is proposed that FTQ data can be used to form empirical distribution functions (EDFs) for sampling in order to drive the Chama simulator noise studies. Examining the EDF corresponding to each of the time series shown in Fig. 6.1, we can see the most

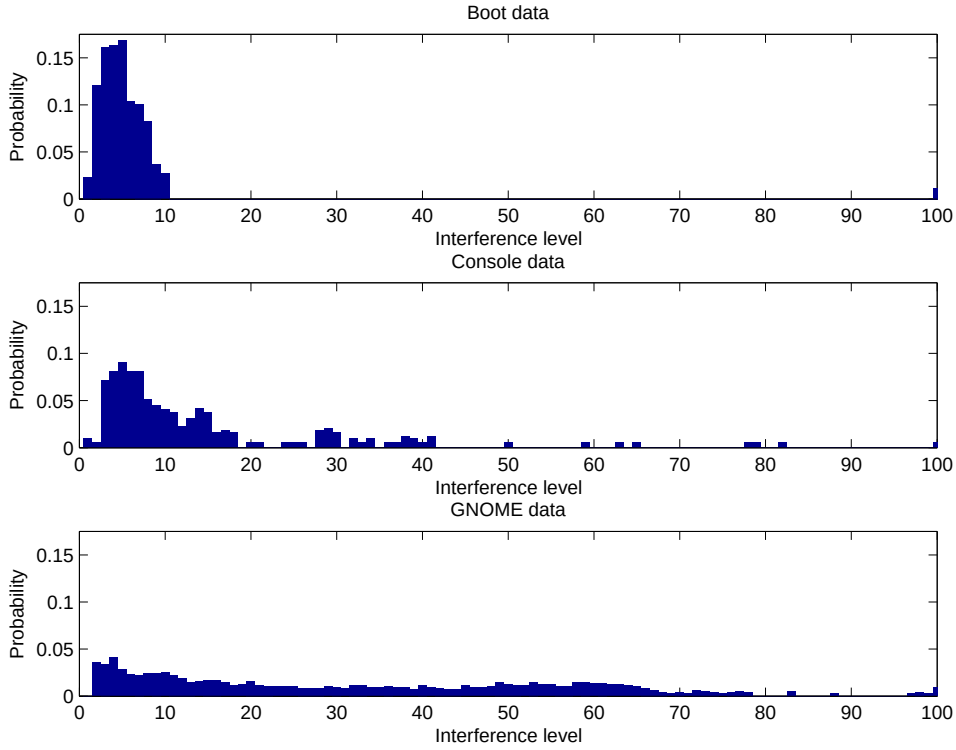

Figure 6.2: Three empirical distribution functions derived from data in Fig. 6.1.

compelling evidence that the stochastic nature of the noise present in each sample varies drastically. The EDFs for each data set are shown in Fig. 6.2.

The EDFs show that as noise increases, the probability of a low impact perturbation decreases. The pre-boot measurement shows that most samples correspond to low perturbation events, while the highest noise case shows that moderate to high impact perturbations are equally likely as low impact perturbations. Furthermore, these give additional evidence that to accurately model the noise within a system, sampling from an idealized distribution function is inappropriate.

FTQ work quantum variation

Due to the small number of instructions that comprise the FTQ benchmark, it is easy to allow artifacts from processor scheduling and instruction mix to overwhelm the coarser-grained noise that FTQ seeks to measure. The effect of varying the work quantum granularity of FTQ is demonstrated in this section, and was introduced earlier in Section 4.4. In these experiments, the level of noise was fixed, corresponding to the post-boot environment on an X41 laptop without any GUI environment started. This represents a relatively low noise, full operating system configuration. Four granularities of FTQ work quantum were tested: one integer increment operation, 15, 31, and 63 integer operations. The raw FTQ data from these experiments is shown in Fig. 6.3.

The data illustrates two important effects from varying the work quantum granularity. First, as the work quantum becomes coarser, the number that can be executed in a fixed sampling period decreases. Furthermore, this decrease is approximately proportional to the amount the workload increases. For a single increment work quantum, the mean work per sample is approximately 15167 units. For a 15 operations per quantum, this mean work per sample decreases to 5267 units, with a similar decrease to 3147 units for 31 operations and 1753 units for 63 operations. Most importantly, the variance of each time series decreases drastically, while preserving significant perturbations as visible downward spikes in the raw data. The standard deviation (or $\sqrt{variance}$) decreases from 343 units in the single instruction case progressively through values of 49, 39, and 19 units as the work quantum granularity increases. By coarsening the work quantum and increasing the proportion of work instructions per sample versus control flow instructions, we are able to smooth out self-interference due to low level hardware effects within the processor.

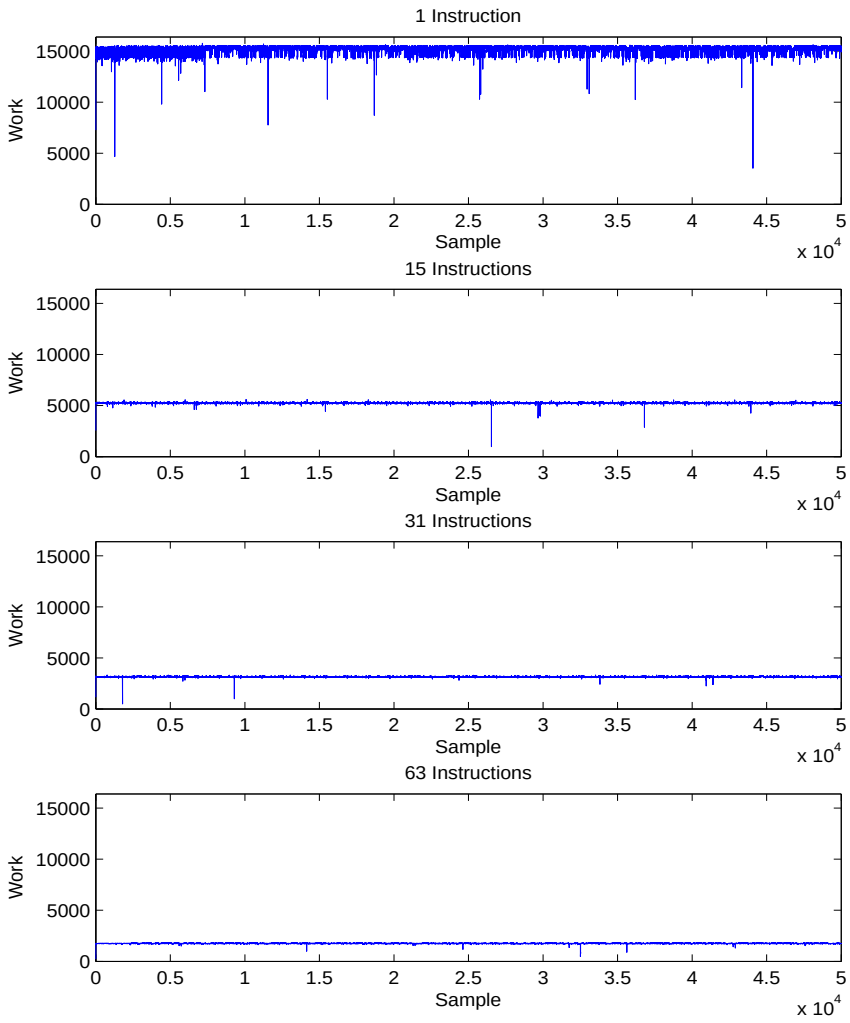

Figure 6.3: Four FTQ time series representing increasing granularity work quanta.

It must be noted that coarsening requires sacrificing fidelity in the data. As the work quantum increases in granularity, the ability to differentiate large perturbations from small ones decreases. All perturbations less than or equal to the scale of a single work quantum are indistinguishable from each other. This sacrifice is acceptable in the context of this work. We are simply seeking a method to identify periodic, high-impact perturbations due to preemption for other operating system tasks. We do not currently wish to identify the magnitude of these perturbations.

6.3 Experimental setup for Chama studies

The traces used to drive the Chama studies were measured on a 24 processor, 12 node cluster of dual processor, 2.8Ghz Pentium 4 Xeon based nodes connected with Gigabit Ethernet. The codes were compiled with the Intel C and Fortran compiler version 8.1. The OpenMPI version 1.0.1 implementation of MPI was used for message-passing support. Traces were taken with the PMPI-based tracing library provided with Chama.

Each code was executed once to completion under normal conditions using the provided test inputs used for benchmarking and testing the codes. The FLASH and Sweep3d codes were executed using 24 processors, and the NAS codes were executed in their Class B configuration on 16 processors due to the constraint that a power of two processors were required.

We ran the Chama simulator with two parameter settings. The first configuration is provided a probability of perturbation on each processor over a period of 10ms (or 28,000,000 ticks). We varied these probabilities from 0.0 to 0.9 by increments of 0.02. This configuration represents the situation where noise is increased in intensity across the entire cluster, as would be the case in a homogeneous configuration of node operating systems. The second configuration is similar, but only one node

experiences perturbations. This represents the case where a single node in the cluster is misconfigured.

Each data point represents an average value for a set of simulation runs with identical parameters and inputs, varying only the random number seed used to sample for simulated delay injection. This set is composed of 20 simulation runs for each probability level and simulation code. With the original Chama code, this simulation count is chosen based on the simulation performance. Ideally, after optimization of the simulation it will be possible to increase the run count and better capture the expected statistics (mean and variance) for a single parameter set. The design of a simulation-based experiment must take into account these sorts of statistical factors in order to generate sound results with a reasonable number of simulation runs [42].

The hypothesis that this experimental study tests has two components.

1. As the probability of noise increases, the amount of noise that cannot be absorbed by the program increases.

2. The increase in runtime experienced by programs should be higher when noise is present on all nodes, versus a single node. Furthermore, codes that rely solely on synchronous operations will experience similar slowdowns for both the whole cluster and single node perturbation cases.

Details that are specific to each code are provided in the corresponding sections below.

6.3.1 Delay propagation details

Delay is propagated using the methods described in Chapter 5. The exact manner by which delay is propagated must be discussed, as other methods can be used which

are equally valid under differing sets of assumptions. In this study, the duration of events was not modified in all but the clearest case of a paired set of synchronous point-to-point message transfer operations. If an `MPI_Send` operation took a specific amount of time, this duration was not modified if the receive was asynchronous. The reason for this is that the duration of the paired receive and corresponding `MPI_Wait` call contains no information about the proportion of time spent idle in the send waiting for the receive to be posted versus the amount of time required to transfer the message. The only time this inference is reasonable is within the paired `MPI_Send`/`MPI_Recv` operations, where the minimum duration of the pair can be assumed to be the time for the transfer.

A more aggressive set of assumptions can use knowledge of message sizes and the time required for a fixed amount of data to transfer in order to shrink operations and better absorb injected delay. Such details may also depend on the MPI implementation used by the application. Although Chama contains information for each message related to the precise message size, no information is currently integrated into the delay propagation scheme that allows Chama to use this to compute the expected time for a message versus the observed and simulated times in order to recompute event durations. The codes that are highly dependent on mixed syncronous/asynchronous messages may experience better noise absorption than presented here given a more sophisticated delay propagation model.

6.4 Sweep3d

The Sweep3d benchmark [36] was the simplest code examined due to its small size and low number of message passing primitives used. A trace of the code reveals that it alternates between phases of synchronous two-sided communications and collective operations. Table 6.4 lists the MPI operations used by the Sweep3d code. Due to its

MPI Operation	Percentage	Avg. Exclusive Time (ticks)
MPI_Allreduce	1.04	11773650
MPI_Barrier	0.10	6777461
MPI_Bcast	0.13	9076706
MPI_Recv	49.37	853294
MPI_Send	49.37	190542

Table 6.4: MPI operations used by Sweep3d

dependence on purely synchronous operations, it is easy to hypothesize before simulation that any noise injected into the system would result in increases in runtime. Without asynchronous sends or receives traditionally used to make an application latency or perturbation tolerant, any perturbation is likely to be seen in the overall runtime unless the processors are not load balanced, and the perturbations are absorbed by idle periods spent by some processors at collective operations (refer to Fig. 1.1 for an illustration of this arrival time skew issue).

The trace data was taken for a 24 processor run using the 50x50x50 input deck provided with the Sweep3d code download package using a 6 by 4 processor grid.

As we can see in Fig. 6.4, the sweep code is able to tolerate delay at the full range of magnitudes equivalently. Furthermore, the sweep code can absorb delay better if it originates from a single process than uniformly over all processes. Delay that is induced across all compute nodes cannot be completely absorbed before it propagates across all processes. Single node delay can be absorbed, although as the rate of delay increases, it becomes more difficult for single processor delays to be absorbed before propagating across the entire program.

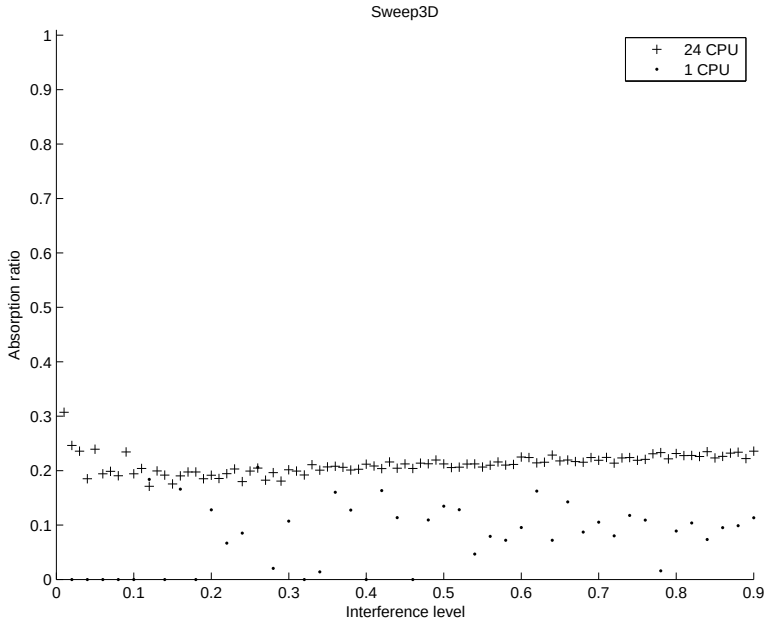

Figure 6.4: Absorption ratio for a 24 CPU Sweep3d trace under increasing interference levels.

6.5 The NAS Parallel Benchmarks

Three codes from the MPI implementation of the NAS parallel benchmark suite version 3.2 were chosen for study: the CG (Conjugate Gradient), LU (LU decomposition based CFD), and MG (Multigrid) benchmarks. Each was compiled in its Class B configuration for 16 CPUs. The MPI operation mix varied between these codes from predominantly synchronous point-to-point in the LU case, to a mix of synchronous sends with asynchronous receives in the MG and CG case. Collectives formed a small fraction of the operations executed in each case. The operation mix for all three codes is shown in Table 6.5.

In Fig. 6.5, the ratio of noise not absorbed versus total noise experienced is shown for each code. The MG and CG codes do not absorb noise well at any level, regardless

	MPI Operation	Percentage	Avg. Exclusive Time (ticks)
CG	MPI_Barrier	<0.01	8949093
	MPI_Irecv	33.33	46361
	MPI_Send	33.33	4703508
	MPI_Wait	33.33	4658358
MG	MPI_Allreduce	1.03	14502693
	MPI_Barrier	0.07	38458045
	MPI_Bcast	0.07	15341284
	MPI_Irecv	32.94	41886
	MPI_Reduce	0.01	267138
	MPI_Send	32.94	3184664
	MPI_Wait	32.94	2550534
LU	MPI_Allreduce	0.01	101188654
	MPI_Barrier	<0.01	2960303
	MPI_Bcast	0.01	12894221
	MPI_Irecv	0.50	85453
	MPI_Recv	49.24	1732421
	MPI_Send	49.74	598220
	MPI_Wait	0.50	36345911

Table 6.5: MPI operations used by NAS CG, MG, and LU.

of if the noise is experienced on only one process or uniformly across all processes. The LU code on the other hand shows less tolerance for noise when it is experienced on all processors versus the case when noise occurs only on one. In both cases, the LU code appears to have better noise tolerance characteristics than the MG or CG codes.

This is an interesting result. The standard assumption is that the use of asynchronous operations (such as MPI_Irecv) allows codes to be more tolerant of delays induced by parameters such as interconnect latency. Although MG and CG rely heavily on these, it appears they are not having their desired effect. Although no deeper investigation of the delay propagation at the event-level has been made, one can look at the exclusive time data in Table 6.5 and see a potential reason for this

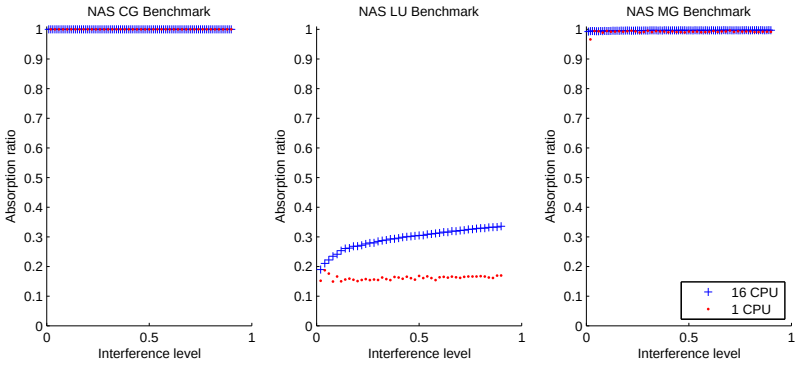

Figure 6.5: Absorption ratio for 16 CPU traces of the Class B NAS CG, LU, and MG codes under noise on all processes and a single process.

behavior. In the MG and CG case, the synchronous operations that were paired with the asynchronous operations have approximately the same exclusive time as the `MPI_Wait` operation that blocks pending completion of the synchronous counterpart. This can mean that the wait operation is not spinning idle for long, and is occurring at approximately the same time as the synchronous operation. The traces leave little room to shrink the duration of the wait under delay.

On the other hand, the LU case shows a very large (at least one order of magnitude) difference in the time for a wait to complete versus the synchronous counterpart. This means that the wait operations are able to shrink and absorb a great deal of delay locally due to the late arrival in the trace of the process executing the synchronous operation. This shows that the instruction mix does not dominate the absorption characteristics of the code. The natural load imbalance seen at runtime appears to have a stronger effect on absorption.

6.6 ASCI FLASH

The ASCI FLASH code is a parallel simulation for studying astrophysical thermonuclear flashes developed at the University of Chicago. This code was chosen to represent a highly complex parallel program that contains many different phases of execution. The FLASH code alternates between phases of numerical computation on a mesh, adaptive mesh refinement using the PARAMESH [39] package, and checkpointing I/O operations. FLASH is, from a parallel message passing perspective, very representative of large workloads that consume time on large scale parallel machines throughout the scientific community. As such, the FLASH code had two benefits for this work. First, the non-trivial nature of its message passing graph was a difficult test for the Chama code. It was used extensively in the development and debugging process. Second, its irregularity causes it to have different noise absorption characteristics at different points in its execution.

In addition to having a richer structure than other codes, it also possesses a wider coverage of MPI operations. Table 6.6 gives the set of operations that it uses along with the instruction mix breakdown and the exclusive time spend within each operation.

The FLASH code shows poor absorption of simulated noise. The use of collective operations between phases of computation appears to degrade the ability of the code to tolerate noise. The FLASH code alternates between phases of computation based on pairwise operations separated by collective operations. Any delay that was induced over a subset of processes and not absorbed immediately propagates to all processors at this time. Although not shown here, it is possible to watch the execution of the code under the Chama simulator, and see delay displayed for each process in the GUI build up on subsets of processes only to suddenly propagate to all processes as soon as a collective occurs.

MPI Operation	Percentage	Avg. Exclusive Time (ticks)
MPI_Allgather	0.02	46215857
MPI_Allreduce	1.58	15249390
MPI_Barrier	1.00	131564619
MPI_Bcast	2.12	11141296
MPI_Gather	0.02	184011
MPI_Iprobe	0.12	71427
MPI_Irecv	42.34	10871
MPI_Isend	0.04	50382
MPI_Recv	0.16	658340
MPI_Reduce	0.30	348222
MPI_Scan	0.06	5193525
MPI_Send	0.12	502065
MPI_Sendrecv	0.10	2566739
MPI_Ssend	42.34	2457312
MPI_Waitall	9.69	8173038

Table 6.6: MPI operations used by ASCI FLASH

6.7 Comparison of codes

The most striking results of the initial Chama simulations are apparent when comparing the noise absorption characteristics of multiple codes under equal noise environments. Studying how multiple codes react to the same level of noise is important. If a set of codes has been run on a single platform to generate traces under similar noise conditions, these traces can be used to evaluate different platforms to study which codes will experience the most performance degradation. If some codes absorb noise better than others in the new noise environment, then they are better suited to running in this new environment than others. Their performance will be closer to their original performance than codes that fail to absorb additional noise as well.

Three characteristics were most important in differentiating the different codes. These were the mix of MPI operations, the amount of time required for the code to

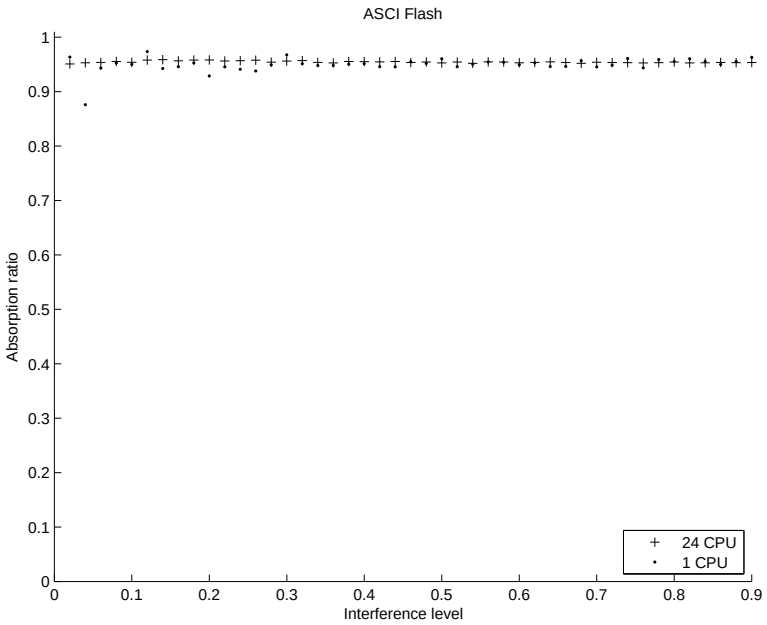

Figure 6.6: Absorption ratio for a 24 CPU FLASH trace with simulated delays on all processes and a single process.

complete, and the amount of time spent in local computation between MPI operations. The mix of MPI operations has an impact on whether or not noise absorption is a function of inherent load imbalance alone, or if noise-tolerant, asynchronous operations can assist. The amount of time required to complete the code dictates the amount of noise experienced by a program, as the noise level is defines as noise per 10 ms of wall clock time regardless of the runtime of the program. Finally, the amount of time spent in local computation between MPI operations dictates the likelihood of a delay being experienced between any two consecutive calls.

Using the Chama output shown in the previous sections, we now make a comparison of these codes. The first comparison shown in Fig. 6.7 shows the amount of delay (in CPU ticks) absorbed by the codes under increasing amounts of interference on a single process in the set of parallel processes. The absorbed delay is shown on

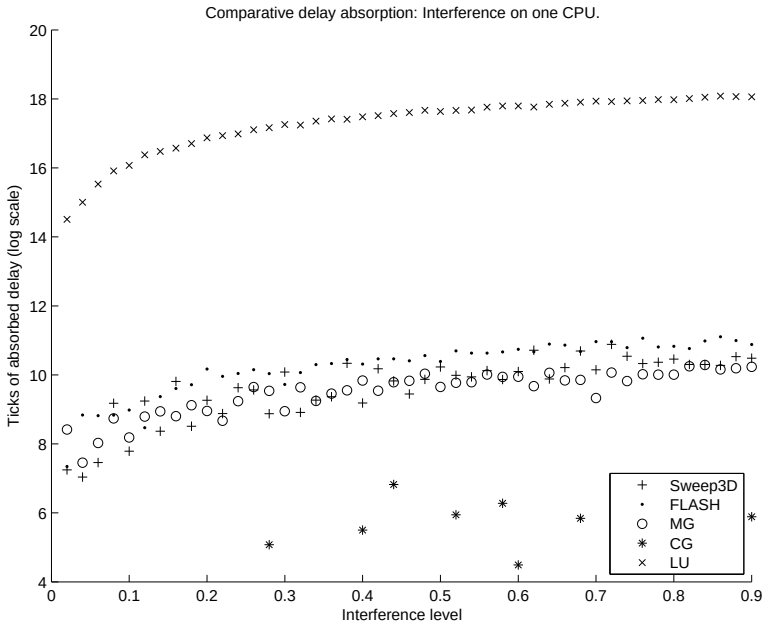

Figure 6.7: A comparison of absorption of noise injected on one processor for the Sweep3d, FLASH, and NAS codes.

a logarithmic scale, while the interference level is a linear scale. The NAS LU code shows the greatest interference absorption, while the NAS CG code shows the least. The FLASH, NAS MG, and Sweep3d codes show comparable absorption characteristics. This plot shows that, although the absorption ratios shown in previous sections make it difficult to see absorption by three of the codes, they do in fact absorb noise, and do so to differing degrees.

A second comparison is shown in Fig. 6.8, again using a logarithmic scale for the total absorbed delay and a linear scale for the level of interference. In this case, interference was injected into all processes participating in the parallel job instead of only one. This means that the amount of delay experienced by the parallel program as a whole was much higher. Furthermore, it means that processes that were forced to be idle in the case with a single slow process are also delayed, reducing the time

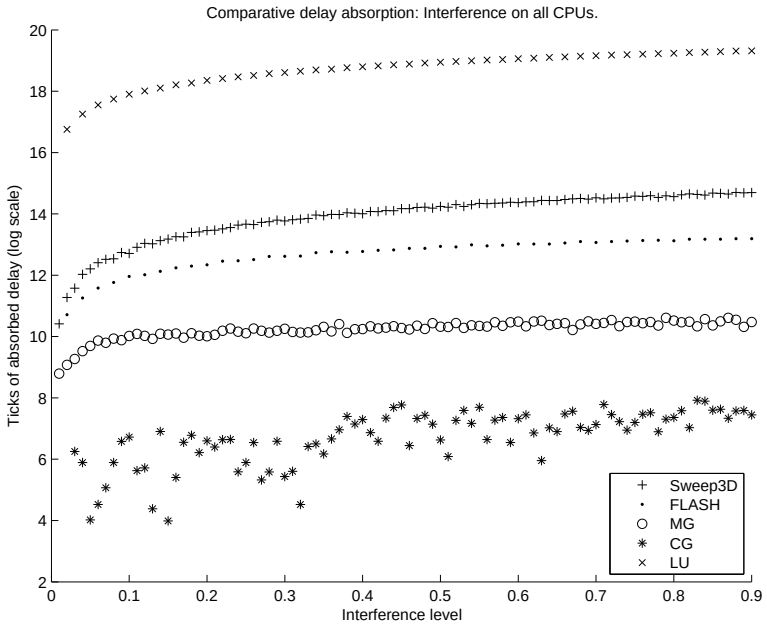

Figure 6.8: A comparison of absorption of noise injected on all processors for the Sweep3d, FLASH, and NAS codes.

they spend idle. As the data shows, the amount of delay that can be absorbed by each code differentiates more. The NAS LU code remains the best at absorbing delay, while the NAS CG code remains the least capable of noise absorption. The interesting part of this comparison with respect to the single processor noise injection case is that the FLASH, Sweep3d, and NAS MG codes now show different levels of noise absorption.

It is very likely that in future development of the Chama simulator, relaxing the conservative method by which delay is propagated through asynchronous operations and the corresponding `MPI_Wait` calls will allow additional absorption by codes that utilize them. The conservative implementation used for these experiments does allow some degree of absorption as the data shows, but not at the level it potentially can.

Chapter 7

Conclusion

In this dissertation, the performance analysis practice for studying the effect of interference parallel computers has been defined as a mix of detailed systems benchmarking and application interference simulation. The use of well defined tools, microbenchmarks, for inferring machine performance characteristics was described and demonstrated. This assists in quantifying the effect of operating system and process preemption perturbations on the performance of individual computational nodes in parallel programs. Application traces representing real execution behavior of parallel programs are used to simulate their modified behavior under synthetic performance conditions to explore application specific sensitivities to performance parameter variations. The data quantified by microbenchmarks is then used to parameterize these application sensitivity studies.

This process of measurement of machines and applications, followed by a synthesis of this information to analyze application performance, forms the core of the methodology. This dissertation has provided new techniques that address the following three goals:

1. A tool for quantifying operating system interference through the FTQ mi-

crobenchmark.

2. A trace-driven simulation for performing analysis of application sensitivity to performance perturbations.

3. Demonstration of these tools in quantifying interference on real platforms and its impact on real applications.

This work has already had an impact on systems development at the Los Alamos National Laboratory. The FTQ microbenchmark is routinely used in the Advanced Computing Laboratory to compare noise characteristics on new operating systems research for future generation cluster platforms. The concepts behind FTQ have also been used by others to build similar microbenchmarks for examining operating system interference [6]. Chama also forms a major part of a DOE Office of Science funded FastOS project at Los Alamos. It will help bring a scientific discipline to the quantitative evaluation of parallel programs with respect to their sensitivity to performance perturbations.

7.1 Future work

The work presented in this thesis leaves many areas open to future investigation. These include:

- **Parallel languages:** Many parallel languages, including ZPL [68] and Co-Array Fortran [55], compile high-level representations of parallel programs into a machine-generated sequence of message passing operations. In some cases, the compiler back-end will even produce C code with MPI-based communications. Instrumentation can be inserted into the generated code by the compiler

to produce traces compatible with the simulation tool presented in this thesis. Work that was recently published [71] regarding collective primitives in Co-Array Fortran would benefit from a performance study based on the work presented here.

- **Microbenchmark baselining:** The ability to measure "ground truth" about a parallel platform is vital for decoupling the effects of peripheral devices and basic kernel software from data gathered by the FTQ microbenchmark. A method for using FTQ to create a baseline measurement was discussed in chapter 4. It remains to be investigated how to use these baseline measurements to subtract intrinsic noise from the machine from post-boot FTQ measurements. The inherent error-bars present in the sample intervals due to self-sampling make it impossible to simply subtract spectra without compensating for this sampling interval jitter first.

- **Extension of Chama beyond MPI:** Although Chama is structured to not rely solely on the abstractions found in the MPI standard, it has not yet been used with data from other message passing libraries. The most likely candidate for investigation is the ARMCI library, which is used by the NWChem package [34]. Although MPI is available on nearly all modern parallel computers, other models for message passing may be more appropriate for new and future parallel architectures.

- **Correlation with code:** Currently Chama can provide statistics on the delay absorbed and experienced by classes of operations. It is unable to correlate these delays with specific instances of these operations within the source code to identify regions that should be focused on for optimization. Using tools such as TAU that allow operations to be both traced and tagged with their location in source code, it will be possible to provide such data.

- **Verification and validation:** Most simulation requires some degree of val-

idation and verification to ensure that the simulation both models what the user wants and does so correctly. In simulation based scalability studies, the predictions can be verified by running codes on platforms of the scale modeled. In sensitivity studies, one must artificially induce noise on a system at the level modeled. Constructing and testing hypotheses with such artificial "perturbers" will be an interesting step towards verifying that the simulated results accurately predict noise sensitivity.

References

[1] TOP500 Supercomputer Sites. http://www.top500.org/.

[2] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[3] David A. Bader and Kamesh Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In *Proc. 12th International Conference on High Performance Computing (HiPC 2005)*, volume 3769 of *Lecture Notes In Computer Science*, pages 465–476, December 2005.

[4] Rosa M. Badia, Jesús Labarta, Judit Giménez, and Francesc Escalé. DIMEMAS: Predicting MPI applications behavior in Grid environments. *Workshop on Grid Applications and Programming Tools, 8th Global Grid Forum (GGF8), Seattle, WA*, 2003.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[6] Pete Beckman and Susan Coghlan. ZeptoOS: Small Linux for Big Computers. In *June 2005, FastOS PI Meeting*, 2005.

[7] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[8] George E. P. Box, Gwilym M. Jenkins, and Gregory C. Reinsel. *Time Series Analysis: Forecasting and Control, Third Edition*. Prentice Hall, 1994.

[9] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 42, Washington, DC, USA, 2000. IEEE Computer Society.

[10] Arthur W. Burks, Herman H. Goldstine, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. *U.S. Army Ordnance Department Report*, 1946.

[11] Kris Buytaert. The openMosix HOWTO. *http://howto.ipng.be/openMosix-HOWTO*, 2002.

[12] David Callahan, John Cocke, and Ken Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, 1988.

[13] C. Clos. A study of non-blocking switching networks. *Bell Systems Technical Journal*, 32(2):406–424, March 1953.

[14] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, April 1965.

[15] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, second edition edition, 1994.

[16] Jack Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report CS-89-85, University of Tennessee, Knoxville TN, 37996, 2005.

[17] Jack Dongarra and Piotr Luszczek. Introduction to the HPC Challenge Benchmark Suite. Technical Report UT-CS-05-544, University of Tennessee, 2005.

[18] Jack Dongarra, Allen D. Malony, Shirley Moore, Philip Mucci, and Sameer Shende. Performance Instrumentation and Measurement for Terascale Systems. In *Proc. International Conference on Computational Science (ICCS 2003)*, LNCS 2660, pages 53–62. Springer, 2003.

[19] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: Past, Present, and Future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[20] R. P. Draves, B. N. Bershad, and A. F. Forin. Using microbenchmarks to evaluate system performance. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 148–153, 1992.

[21] Domenico Ferrari. *Computer Systems Performance Evaluation*. Prentice Hall, 1978.

[22] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[23] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modelling Astrophysical Thermonuclear Flashes. *Astrophysical Journal Supplement*, 131:273–334, 2000.

[24] Brian R. Gaeke, Parry Husbands, Xiaoye S. Li, Leonid Oliker, Katherine A. Yelick, and Rupak Biswas. Memory-Intensive Benchmarks: IRAM vs. Cache-Based Machines. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, April 2002.

[25] S. Girona, Jesús Labarta, and Rosa M. Badia. Validation of Dimemas communication model for MPI collective operations. *Proc. 7th European PVM/MPI Users' Group Meeting, Lake Balaton, Hungary*, pages 39, 46, 2000.

[26] Calvin C. Gotlieb and John K. Metzger. Trace driven analysis of a batch processing system. In *Proceedings of the 1st symposium on Simulation of computer systems*, pages 214–222, 1973.

[27] Erik Hendriks. BProc: The Beowulf Distributed Process Space. *16th Annual ACM International Conference on Supercomputing*, 2002.

[28] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with upshot. Technical Report ANL-91/15, Argonne National Laboratory, 1991.

[29] Roger Hockney and Michael Berry. Public international benchmarks for parallel computers report. Technical Report 1, Parkbench Committee, 1994.

[30] Roger W. Hockney. *The Science of Computer Benchmarking*. SIAM, 1996.

[31] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, 1991.

[32] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

[33] E. O. Joslin. Application Benchmarks: The Key to Meaningful Computer Evaluations. In *Proceedings of the 20th ACM National Conference*, pages 27–37, 1965.

[34] Ricky A. Kendall, Edoardo Aprà, David E. Bernholdt, Eric J. Bylaska, Michel Dupuis, George I. Fann, Robert J. Harrison, Jialin Ju, Jeffrey A. Nichols, Jarek Nieplocha, T.P. Straatsma, Theresa L. Windus, and Adrian T. Wong. High performance computational chemistry: An overview of NWChem a distributed parallel application. *J. Comp. Phys. Commun.*, 128(1–2):260–283, 2000.

[35] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions*, EC-11(2):223–235, 1962.

[36] K. R. Koch, R. S. Baker, and R. E. Alcouffe. Solution of the first-order form of the 3-D discrete ordinates equation on a massively parallel processor. *Transactions of the American Nuclear Society*, 65:198–199, 1992.

[37] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, second edition, 1991.

[38] J. S. Liptay. Structural aspects of the system/360 Model 85, part II: The cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[39] Peter MacNeice, Kevin M. Olson, Clark Mobarry, Rosalinda deFainchtein, and Charles Packer. PARAMESH : A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126:330–354, 2000.

[40] Allen D. Malony and Sameer Shende. Performance Technology for Complex Parallel and Distributed Systems. In *Quality of Parallel and Distributed Programs and Systems*, pages 25–41. Nova Science Publishers, Inc., 2003.

[41] John D. McCalpin. A survey of memory bandwidth and machine balance in current high performance computers. *Newsletter of the IEEE Technical Committee on Computer Architecture (TCCA)*, 1995.

[42] Catherine McGeoch. Analyzing Algorithms by Simulation: Variance Reduction Techniques and Simulation Speedups. *ACM Computing Surveys*, 24(2):195–212, 1992.

[43] F. H. McMahon. The Livermore Fortran Kernels: A Computer Test Of The Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, California, 1986.

[44] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference*, 1996.

[45] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee, 1994.

[46] Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface. Technical report, University of Tennessee, 1996.

[47] Barton P. Miller, Mark D. Callaghan, Joanthan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[48] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[49] Ronald Minnich, James Hendricks, and Dale Webster. The Linux BIOS. In *The Fourth Annual Linux Showcase and Conference*, 2000.

[50] B. Mohr, K. Shanmugam, and A. Malony. Speedy: An Integrated Performance Extrapolation Tool for pC++ Programs. In H. Beilner and Falko Bause, editors, *TOOLS95*, pages 254–268. Springer-Verlag, LNCS 977, September 1995.

[51] Bernd Mohr and Felix Wolf. KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, Klagenfurt, Austria, September 2003.

[52] Ronald Mraz. Reducing the variance of point to point transfers in the IBM 9076 parallel computer. In *Proceedings of the 1994 conference on Supercomputing*, 1994.

[53] Myricom, Inc. Guide to Myrinet-2000 Switches and Switch Networks, August 2001.

[54] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP '99*, April 1999.

[55] Robert W. Numrich and John K. Reid. Co-Array Fortran for Parallel Programming. *ACM Fortran Forum*, 17(2):1–31, 1998.

[56] Alan V. Oppenheim, Ronald W. Schager, and John R. Buck. *Discrete-Time Signal Processing, Second Edition*. Prentice Hall, 1999.

[57] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals and Systems, Second Edition*. Prentice Hall, 1997.

[58] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., 1997.

[59] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *ACM/IEEE SC2003*, Phoenix, Arizona, November 10–16 2003.

[60] D. Reed, R. Aydt, T. Madhyastha, R. Noe, K. Shields, and B. Schwartz. An overview of the Pablo performance analysis environment. Technical report, University of Illinois, Urbana, Illinois, 1992.

[61] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.

[62] Giuseppe Rodriguez, Rosa M. Badia, and Jesús Labarta. Generation of Simple Analytical Models for Message Passing Applications. *Proc. 10th Int'l Euro-Par Conference on Parallel Processing, Pisa, Italy*, 2004.

[63] K. Shanmugam and A. Malony. Performance Extrapolation of Parallel Programs. In *Proceedings of ICPP 1995*, pages 117–120, August 1995.

[64] Stephen Sherman. Trace driven modeling: An update. In *Proceedings of the 4th symposium on Simulation of computer systems*, pages 87–91, 1976.

[65] Stephen Sherman and J. C. Browne. Trace driven modeling: Review and overview. In *Proceedings of the 1st symposium on Simulation of computer systems*, pages 200–207, 1973.

[66] Alan J. Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[67] Marc Snir and David A. Bader. A Framework for Measuring Supercomputer Productivity. *International Journal of High Performance Computing Applications*, 18(4):417–432, 2004.

[68] Lawrence Snyder. *Programming Guide to ZPL*. MIT Press, Cambridge, MA, USA, 1999.

[69] Matthew Sottile and Ronald Minnich. Analysis of Microbenchmarks for the Performance Tuning of Clusters. In *Proceedings of Cluster 2004*, 2004.

[70] Matthew J. Sottile, Vaddadi P. Chandu, and David A. Bader. Performance analysis of parallel programs via message-passing graph traversal. In *Proceedings of the 2006 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.

[71] Matthew J. Sottile, Craig E Rasmussen, and Richard L. Graham. Co-Array Collectives: Refined Semantics for Co-Array Fortran. In *Proceedings of the 3rd International Workshop on the Practical Aspects of High-level Parallel Programming (PAPP 2006)*, 2006.

[72] Carl Staelin and Larry McVoy. mhz: Anatomy of a micro-benchmark. In *Proceedings of USENIX technical conference*, 1998.

[73] Henry Stark and John W. Woods. *Probability and Random Processes with Applications to Signal Processing*. Prentice Hall, 2002.

[74] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[75] Vampir web page. http://www.pallas.com/e/products/index.htm.

[76] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001.

[77] M. V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, EC-14(2):270–271, 1965.